Arctic: Lightweight and Stateless Threshold Schnorr Signatures

Chelsea Komlo and Ian Goldberg

University of Waterloo
ckomlo@uwaterloo.ca, iang@uwaterloo.ca

Abstract. Threshold Schnorr signatures are seeing increased adoption in practice, and offer practical defenses against single points of failure. However, one challenge with existing randomized threshold Schnorr signature schemes is that signers must carefully maintain secret state across signing rounds, while also ensuring that state is deleted after a signing session is completed. Failure to do so will result in a fatal key-recovery attack by re-use of nonces.

While deterministic threshold Schnorr signatures that mitigate this issue exist in the literature, all prior schemes incur high complexity and performance overhead in comparison to their randomized equivalents. In this work, we seek the best of both worlds; a deterministic and stateless threshold Schnorr signature scheme that is also simple and efficient.

Towards this goal, we present Arctic, a lightweight two-round threshold Schnorr signature that is deterministic, and therefore does not require participants to maintain state between signing rounds. As a building block, we formalize the notion of a Verifiable Pseudorandom Secret Sharing (VPSS) scheme, and define Shine, an efficient VPSS construction. Shine is secure when the total number of participants is at least $2t - 1$ and the adversary is assumed to corrupt at most $t - 1$; i.e., in the honest majority model.

We prove that Arctic is secure under the discrete logarithm assumption in the random oracle model, similarly assuming at minimum $2t - 1$ number of signers and a corruption threshold of at most $t - 1$. For moderately sized groups (i.e., when $n \leq 20$), Arctic is more than an order of magnitude more efficient than prior deterministic threshold Schnorr signatures in the literature. For small groups where $n \leq 10$, Arctic is three orders of magnitude more efficient.

Table of Contents

1 Introduction

Threshold signature schemes allow a subset of at least μ out of n possible parties to cooperate to produce a signature over a single message, while preserving security when up to $t-1$ signers may be corrupted, where $\mu \leq n$ and $t \leq \mu$. Threshold signatures offer practical security benefits, allowing for dynamic key management and defense in depth against potential adversarial corruptions. While threshold signatures can be instantiated by simply concatenating multiple individual signatures into a single joint signature, a practical goal for threshold signatures is to preserve compatibility with existing single-party signature schemes, to minimize implementation complexity.

In this work, we focus our attention on threshold Schnorr signatures; i.e, a threshold signature scheme that is compatible with the (single-party) Schnorr signature verification algorithm [50]. Efficient (two-round) threshold Schnorr signatures exist in the randomized setting [3,35,49], where each party is assumed to securely maintain state between signing rounds and have access to good sources of randomness. However, efficient and *deterministic* threshold Schnorr signatures has thus far remained an elusive goal.

Why Deterministic Threshold Signatures? Producing threshold signatures in a deterministic manner is useful for two reasons. First, it is useful as a general defense-in-depth measure, to protect against the event of temporarily losing access to good sources of randomness [48], such as if a machine randomly rebooted.

Second, threshold Schnorr signature schemes generally require participants to perform multiple rounds of communication before a joint signature can be issued. As such, participants must keep state between each round, and carefully delete state once a signing protocol is completed. In the setting where the signer must produce signatures at increasing scale and in a concurrent setting, managing state can become a significant performance bottleneck. Further, secure management of secret state can be considered a security risk. In particular, for threshold Schnorr signatures, participants must generate secret nonces for each signing session. If a participant's state is mismanaged in such a way that it is used even twice, a fatal key-recovery attack is possible. However, in the setting where machines can go offline at any time, and signing is done at scale and concurrently, such careful management of secret state becomes even more challenging.

The Challenge of Efficient Deterministic Multi-Party Schnorr Signatures. While the goal of efficient and deterministic multi-party Schnorr signature schemes is desirable, producing such schemes has proven difficult. The challenge can be summarized as follows. While honest participants can certainly pick their nonces deterministically (say, by hashing their secret signing share and the message), a malicious party might deviate from the protocol and pick its nonce at random. Recall that for Schnorr signatures, the output signature $\sigma = (R, z)$ is a commitment R and response z, satisfying the relation $g^z = R \cdot \mathsf{pk}^c$ for a challenge $c \leftarrow H(R, \mathsf{pk}, m)$. In the setting for threshold Schnorr signatures, where (R, z) are contributed to by all signing parties, one party deviating from the protocol results in the challenge c likewise changing. If honest parties cannot verify that other parties followed the protocol, such a deviation would allow an adversary to perform a key recovery attack with as little as two signing queries.

Prior deterministic multi-party Schnorr signature schemes in the literature [24,37,44] can be broken down into two general approaches. We give an overview of the approaches that support the general threshold setting with general (t, μ, n) in Table 1, with more context now.

The first approach requires that signers output zero-knowledge proofs that a Pseudorandom Function (PRF) was evaluated correctly [24,44]. However, this approach adds undesirable performance and complexity overhead. For example, while MuSig-DN [44] requires only two network rounds, proof generation requires approximately 1 second (regardless of the number of signers) for 256-bit security, and verification requires at least $10n$ ms. Similarly, the scheme by Garillot et al. [24] requires three rounds of communication between signers and high complexity and bandwidth overhead.

The second approach by Kondi et al. [37] is to use as a black box a Pseudorandom Correlation Generator (PCG) [11] to generate nonces in a verifiable manner [37]. However, while the PCG used in their construction can support 2-of-2 signing, [37, 45], it remains an open problem as to how to extend such a PCG to the general threshold setting for any t, μ, or n [36]. As such, the scheme by Kondi et al. [37] supports only the $t = \mu = n = 2$ setting.

	Computation	Bandwidth	Num. Rounds	Assumptions	Min. Signers
MuSig-DN [44]	$14210 + 24(n-1)$ group	1189 bytes	2	DDH	n
GKMN21 [24]	$132,000(t-1)$ AES $256(t-1)$ field	$1.01(t-1)$ MB	3	PRF	t
Arctic (this work)	$2t^2 - t + 2$ group $2\binom{n-1}{t-1}$ field, $2\binom{n-1}{t-1}$ hash	97 bytes	2	DL	$2t-1$

Table 1: Comparison of multi-party deterministic Schnorr signature schemes in the random oracle model (ROM) that support general choices of n (i.e., we exclude 2-of-2 schemes). Here, n denotes the number of total possible signers, t denotes the corruption threshold. Computation is given with respect to the operations that dominate; estimates for MuSig-DN are given with respect to a 256-bit elliptic curve; more information is given in Appendix A. Bandwidth similarly is given with respect to a 256-bit elliptic curve [24, 44].

Therefore in this work, we address the following question:

Can we design a simple, deterministic, and stateless two-round threshold Schnorr signature scheme, with similar efficiency to existing randomized schemes, and acceptable security assumptions in practice?

1.1 Our Results

We answer the above question in the affirmative. In particular, we present Arctic, a deterministic and stateless threshold Schnorr signature scheme. For moderately sized groups (i.e., when $n \leq 20$), Arctic is more than an order of magnitude more efficient than existing deterministic schemes in the literature; it is three orders of magnitude more efficient when $n \leq 10$. For even larger groups, Arctic scales linearly relative to the number of processing cores available to most machines. Signers in Arctic are required to generate and maintain secret keys, but otherwise do not need to manage secret state.

To achieve efficient determinism, Arctic requires two tradeoffs. The first tradeoff is in the number of required signers; Arctic requires $\mu \geq 2t - 1$ parties to participate in signing. This requirement is because Arctic assumes the honest majority model. We discuss in Section 1.2 why the assumption of an honest majority is acceptable for some real-world applications.

The second tradeoff is in the required overhead as the signing set grows large. Under the hood, Arctic employs a replicated secret sharing scheme [32], and so requires participants to store a secret key of size $\binom{n-1}{t-1}$. As such, Arctic outperforms other schemes in the literature for moderately sized groups, but incurs a crossover point as the signing set grows large.

Verifiable Pseudorandom Secret Sharing (VPSS). As a building block, we first formalize a cryptographic primitive that we call a *Verifiable Pseudorandom Secret Sharing (VPSS)* scheme. While pseudorandom secret sharing is a standard notion in the literature, and verifiability of such schemes had been employed implicitly in maliciously secure MPC-based schemes [6], we present a formal definition for a VPSS and give game-based security notions [5], building on existing notions of verifiable random functions in the literature [19, 23, 41, 42].

Intuitively, a pseudorandom secret sharing scheme allows a coalition of players holding pre-established secret shares of a random secret to generate shares of additional pseudorandom secrets. A *verifiable pseudoradnom secret sharing scheme* is simply an extension to allow for public verifiability by collectively verifying the outputs for the set of participants. As such, misbehavior of players in a VPSS can be detected assuming

some threshold of honest participants. As we will see in the case of Arctic, a VPSS allows for more efficiently verifying the correctness of the output of a pseudorandom function that is distributed among a set of players, without requiring each player individually to produce a zero-knowledge proof that it followed the protocol.

We then define a concrete VPSS construction that we call Shine that builds upon the pseudorandom secret sharing scheme by Cramer et al. [16]. We augment the scheme by Cramer et al. to additionally define a verification algorithm that is publicly verifiable even when players only publish commitments to their shares; i.e., it preserves secrecy of players' shares while allowing players to verify that all other players followed the protocol honestly. Shine is efficient for moderately sized groups; concretely, evaluation requires $\binom{n-1}{t-1}$ sub-microsecond hash and field operations, and verification requires $2t^2 - t$ group exponentiations, where t is the corruption threshold. We prove that Shine is secure assuming the existence of a secure hash function in the honest majority model, i,e., when $\mu \geq 2t - 1$, where μ is the minimum number of participants required to participate in the protocol. Additionally, we require that as n grows, t remains $\mathcal{O}(1)$ so that $\binom{n-1}{t-1} = \text{poly}(n)$.

A New Two-Round, Deterministic, and Stateless Threshold Schnorr Signature. Next, we introduce Arctic, a new two-round, deterministic, and stateless threshold Schnorr signature. Arctic is an order of magnitude more efficient than related schemes in the literature for moderately sized groups ($n \leq 20$), due to its use of Shine as a building block. Furthermore, Arctic supports the general threshold setting for any t, μ and n, so long as $\mu \geq 2t - 1$ and $\mu \leq n$. We give an overview of Arctic in Table 1 in comparison to related schemes in the literature.

At a high level, Arctic uses Shine in the first round of signing to generate participant nonces and commitments; participants publish their commitments without having to maintain state of their (secret) nonce. Then, in the second round, participants use Shine to re-derive their nonce and commitment, and verify all other participants' commitments, ensuring that other participants followed the protocol correctly. If the verification check holds, participants derive the group commitment as the aggregation of all participants' individual commitments, and generate a signature share as a combination of their nonce, challenge, and secret signing share. The output signature is the aggregated group commitment and an aggregation of all participants' signature shares, and can be verified using the single-party Schnorr verification algorithm.,

We prove the unforgeability of Arctic assuming the hardness of the discrete logarithm problem in the random oracle model, assuming $\mu \geq 2t - 1$, the adversary corrupts at most $t - 1$ players, and $\binom{n-1}{t-1} = \text{poly}(n)$. We describe in Section 1.2 why requiring honest majority assumptions can be practical for some real-world deployments of threshold signatures, at the benefit of improved simplicity and performance.

Performance Analysis of Arctic. To estimate the practicality of Arctic for various choices of n, t, and μ, we implement the scheme and provide concrete benchmarks.

Arctic is highly efficient for moderately sized groups, and so is a good choice for such settings. As we show in greater detail in Section 6, for a group where $2t - 1 \leq \mu \leq n \leq 20$, Arctic signing operations require less than 100 milliseconds of computation for each signer, and for $n \leq 10$, less than 1 millisecond. Signing however increases in cost relative to the corruption threshold; for signing sets of size $n = 25, t = 11$, the computational overhead for each signer on a single core requires one second. However, Arctic is highly parallelizable, showing almost perfect linear speedups for up to 32 cores for that size of signing set.

For comparison, MuSig-DN requires about 1 second in computational overhead per signer [44], regardless of the size of the signing set. As such, Arctic is a practical choice for moderately sized groups of $n \leq 25$, whereas MuSig-DN or GKMN21 [24] may be good choices as the set of signers grows large, depending on the parallelization of each scheme.

Our Contributions. In summary, the contributions of our work are as follows:

- We formalize the definition of a Verifiable Pseudorandom Secret Sharing (VPSS) scheme, and give game-based security notions.
- We define a concrete VPSS scheme that we call Shine, which extends the pseudorandom scheme by Cramer et al. [16] to allow for public verifiability, assuming an honest majority of participants.

– We then present Arctic, a deterministic and stateless two-round threshold Schnorr signature scheme. Arctic uses Shine as a building block to generate participant nonces and commitments, and verify that participants performed this action correctly.
– We prove that Arctic is secure under the discrete logarithm problem in the random oracle model, assuming Shine is a secure VPSS, the adversary corrupts fewer than t parties, at least $\mu \geq 2t-1$ parties participate in the signing protocol, and $\binom{n-1}{t-1} = \text{poly}(n)$.
– We provide performance benchmarks for Arctic for different sizes of signing sets, and show that parallelization enables a linear speedup in signer computation.

1.2 Observations of Honest Majority Assumptions in Practice

While honest minority assumptions may be desirable for some real-world applications, we observe that honest majority assumptions may be an acceptable tradeoff for other applications in exchange for statelessness, improved performance, and protocol simplicity.

For applications that can easily support additional signers, moving from an honest minority setting to honest majority can be relatively straightforward. For example, an application that currently uses a $(t = 2, \mu = 2, n = 3)$ configuration can instead move to a $(t = 2, \mu = 3, n = 4)$ configuration.

In settings that require liveness, honest majority requirements are already assumed, to ensure usability of the shared secret key even when corrupt players refuse to participate. For example, applications such as cryptocurrency wallets often implicitly require honest majority assumptions, to ensure that corrupt players cannot prevent use of funds by simply refusing to participate in signing operations.

Finally, some applications may see the requirement for additional signers as an acceptable tradeoff to mitigate the security risk of protocol complexity, as protocol complexity increases the risk of exploitable implementation errors [40].

2 Related Work

Randomized (Non-Deterministic) Threshold Schnorr Signatures. We review only threshold Schnorr signatures in the literature that are secure in a concurrent setting and therefore demonstrated to be secure against ROS attacks [7, 20, 51].

Stinson and Strobl [53] present a five-round threshold Schnorr signature that uses a robust DKG [28] for generating its nonce. However, the protocol assumes all participants choose their inputs in a randomized manner.

FROST [35] is a randomized two-round threshold Schnorr signature that is secure even when the first round is preprocessed; i.e, it is performed in a batched manner resulting in only a single online signing round. FROST is secure assuming the One-More Discrete Logarithm (OMDL) assumption in the Random Oracle Model (ROM) [3]. Variants of FROST have been presented to improve its computational and bandwidth efficiency, including FROST2 [3] and FROST3 [14, 49]. Further, an IETF informational draft exists for the original FROST [15]. However, both FROST and subsequent variants rely on participants selecting their nonces at random.

Thee-round randomized threshold Schnorr signatures have likewise been proposed. Lindell [38] presents a three-round threshold Schnorr signature scheme that relies on Fischlin zero-knowledge proofs [22] to demonstrate its simulatability with respect to Schnorr under aborts. Sparkle [17] is a three-round threshold Schnorr signature that is secure assuming the Discrete Logarithm (DL) assumption in the ROM. Sparkle does not rely on heavyweight proofs of knowledge, and instead demonstrates unforgeability via a game-based definition. Makriyannis [39] similarly presents two separate three-round threshold Schnorr signatures, which achieve a similar notion of security as to Sparkle. However, each scheme likewise relies upon randomized nonces.

Deterministic Threshold Signatures. The BLS signature scheme [9] is deterministic, because the signature is of the form $z \leftarrow H(m)^{\mathsf{sk}}$. Likewise, threshold BLS signatures are similarly deterministic [8]. However, such schemes are often not viable in a practical setting that requires backwards compatibility with Schnorr signature verification, or due to the requirement of pairings.

Deterministic threshold Schnorr signatures have been described in the literature, but rely on heavyweight zero-knowledge proofs to demonstrate that each participant followed the protocol correctly. The challenge of deterministic threshold Schnorr signatures is that of *verifiability*, because if even one participant deviates from the protocol and chooses its nonce randomly, a complete key-recovery attack is possible.

Nick et al. [44] define a threshold Schnorr n-of-n multisignature that uses SNARKs to demonstrate in zero-knowledge that a participant generated its nonce using a PRF correctly with respect to a pre-established keypair. However, the authors report the computational overhead of at least 943 ms for a single execution, independent of the number of signers, due to the overhead of proving the PRF was evaluated correctly in zero-knowledge. In particular, to instantiate the PRF, a non-algebraic cryptographic hash function $H : \{0,1\} \to \mathbb{G}$ must be used, along with a regular function $f : \mathbb{G} \to \mathbb{Z}_q$. The complete PRF F_{sk} is defined by $F_{\mathsf{sk}}(x) = f(\mathsf{sk} \cdot H(x))$, where sk is a PRF key. Then, signers generate a Bulletproof [12] to prove in zero-knowledge that the nonce was derived with respect to F, sk, and some input $H(x)$.

Bronte et al. [10] define an honest majority deterministic threshold Schnorr scheme that relies on generic MPC techniques. The authors give two approaches for proving that the PRF was performed correctly. The first approach uses a protocol based on garbled circuits by Hazey et al. [31]; the second uses a replicated secret-sharing approach over $\mathbb{F}_2$, building upon Araki et al. [1].

Garillot et al. [24] similarly rely on parties sampling and committing to a PRF key at the time of key generation, but instead make use of the Zero-Knowledge from Garbled Circuits (ZKGC) paradigm [34] to demonstrate that the PRF was derived correctly. The specific function that is garbled is $C(x) = \phi(F(x))$, where F is a boolean circuit such as AES, and ϕ is standard exponentiation (or curve multiplication). The authors give efficiency estimates for a 256-bit curve and using SHA-512 as the PRF F; the performance overhead is dominated by performing $132,000$ AES invocations and 256 additions in $\mathbb{Z}_q$ per proof generated and verified. For a signing invocation involving t parties, each party must then perform $256t$ field operations and $132,000t$ AES invocations, accounting for each proof a signer generates and the $t-1$ proof verifications for all other signers.

Kondi, Orlandi, and Roy [37] take a different approach, and define a two-round stateless and deterministic two-party Schnorr signature scheme using pseudorandom correlation functions (PCFs) [11] as a building block. In particular, their scheme employs a Paillier-based PRF [45], but additionally define a verification mechanism to ensure that parties honestly followed the protocol. However, their scheme is restricted to the two-party setting, and as written, cannot be extended to the general (t, n) threshold setting. In particular, the PCF used as a building block by their scheme assumes only two parties. Extending their scheme to any (t, n) setting requires designing a new n-party PCF [36].

Honest Majority Threshold Signatures. Honest majority threshold schemes have been proposed in the literature as a means to achieve properties that are either impossible, or require higher performance overhead, in the honest minority setting. Notably, honest majority schemes have been demonstrated to achieve robustness or to circumvent requiring the use of heavyweight zero-knowledge proofs [13].

Gennaro et al. [26] use error correcting codes in the honest majority setting to achieve a robust threshold DSS scheme. Similarly, Ruffing et al. [49] present a wrapper to the FROST threshold signature scheme to achieve robustness, in the honest majority setting.

In the threshold ECDSA setting, Damgård et al. [18] present a threshold ECDSA scheme in the honest majority model, that uses multiplication triples and Beaver's inversion protocol [2].

Distributed VRFs. Galindo et al. [23] give a formalization of distributed VRFs and their security notions. While our notions for a Verifiable Pseudorandom Secret Sharing (VPSS) scheme are similar, our definition for a VPSS does not require that players' outputs are accompanied by a zero-knowledge proof that the protocol was performed correctly. Instead, the VPSS verification function *collectively* verifies all parties' outputs.

Pseudorandom Secret Sharing. Our VPSS construction Shine builds on the pseudorandom secret sharing scheme by Cramer et al. [16]. Note that while Cramer et al. additionally define a Non-Interactive Verifiable Secret-Sharing (NIVSS) variant, their construction assumes that players output shares in the clear, and requires an honest supermajority so that the secret can be recovered. In our case, players verify the correctness of their shares with respect to other players' public commitments, and requires only honest majority (as players simply need to determine if any other party deviated from the protocol).

While we are the first to do so in a threshold Schnorr signature setting, pseudorandom secret sharing has been used as a building block in other other threshold settings. For example, Jarecki, Krawczyk, and Resch [33] define a threshold Oblivious PRF which likewise builds upon pseudorandom secret sharing.

3 Preliminaries

3.1 General Notation

Let $\lambda \in \mathbb{N}$ be a security parameter. We denote the assignment of an element y to the value x as $y \leftarrow x$, and sampling an element from some set S uniformly at random as $x \xleftarrow{\$} S$. For a randomized algorithm A, we write $x \xleftarrow{\$} A()$ to indicate the random variable x that is output from the execution of A.

We use $[n]$ to represent the set $\{1, \ldots, n\}$. For a set S, we denote $\binom{S}{t}$ to mean the set that consists of all size-t subsets of S.

Groups and Group Generation. Let $\mathbb{G}$ be a cyclic group of prime order q, and $\mathbb{Z}_q$ be the field of integers modulo q. Let g be a generator of $\mathbb{G}$, and let $I_{\mathbb{G}} \in \mathbb{G}$ be the identity element of $\mathbb{G}$.

We use $\mathsf{GroupGen}(1^\lambda)$ to denote a polynomial-time algorithm that takes as input a security parameter λ and outputs a group description $(\mathbb{G}, q, g)$.

Polynomial Interpolation. A polynomial of degree $t-1$ over a field $\mathbb{F}$ can be interpolated by t (or more) points. Let η be the list of t distinct indices $\eta \subseteq [n]$ corresponding to the x-coordinates $x_i \in \mathbb{F}, i \in \eta$. Then the $L_i(x)$ (for $i \in \eta$) are the Lagrange polynomials defined by η, of the form

$$L_i(x) = \prod_{j \in \eta; j \neq i} \frac{x - x_j}{x_i - x_j}$$

Later in this work, we denote $L_i(0)$ as λ_i.

Given a set of t points $(x_i, f(x_i))$, any point $f(x_\ell)$ on the degree $t-1$ polynomial f can be determined by Lagrange interpolation:

$$f(x_\ell) = \sum_{j \in \eta} f(x_j) \cdot L_j(x_\ell) \ .$$

3.2 Definitions and Assumptions

Assumption 1 (Discrete Logarithm Assumption (DL)) *The discrete logarithm assumption holds for* $\mathsf{GroupGen}$ *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{A}}(\lambda)$ *is negligible, where*

$$\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{A}}(\lambda) = \Pr\left[\begin{array}{lll} (\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda) & & \\ X \xleftarrow{\$} \mathbb{G} & : & X \stackrel{?}{=} g^{x'} \\ x' \xleftarrow{\$} \mathcal{A}((\mathbb{G}, q, g), X) & & \end{array}\right]$$

Schnorr Signatures. A Schnorr signature is a Sigma protocol zero-knowledge proof of knowledge of the discrete logarithm of the public key, made non-interactive and bound to the message m by the Fiat-Shamir transform [21]. Schnorr signatures are secure under the discrete logarithm assumption in the random oracle model [47].

Definition 1 (Schnorr Signatures [50]). *The Schnorr signature scheme is defined as follows:*

- $\mathsf{Schnorr.Setup}(1^\lambda) \to \mathsf{par}$*: On input the security parameter, run* $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$ *and select a hash function* $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$*. Output public parameters* $\mathsf{par} = ((\mathbb{G}, q, g), \mathsf{H})$ *(which are given implicitly as input to all other algorithms).*
- $\mathsf{Schnorr.KeyGen}() \to (\mathsf{pk}, \mathsf{sk})$*: Sample a secret key* $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ *and compute a public key* $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$*. Output* $(\mathsf{pk}, \mathsf{sk})$*.*
- $\mathsf{Schnorr.Sign}(\mathsf{sk}, m) \to \sigma$*: On input secret key* sk *and message* m*, the signer samples a nonce* $r \xleftarrow{\$} \mathbb{Z}_q$ *and computes a nonce commitment* $R \leftarrow g^r$*. The signer then computes the challenge* $c \leftarrow \mathsf{H}(R, \mathsf{pk}, \mathsf{m})$ *and the response* $z \leftarrow r + cx$*. Output the signature* $\sigma = (R, z)$*.*
- $\mathsf{Schnorr.Verify}(\mathsf{pk}, \mathsf{m}, \sigma) \to 0/1$ *: On input the public key* pk*, a message* m*, and a purported signature* $\sigma = (R, z)$*, the verifier computes* $c \leftarrow \mathsf{H}(R, \mathsf{pk}, \mathsf{m})$ *and accepts if* $g^z = R \cdot \mathsf{pk}^c$*.*

Shamir secret sharing. The (t, n) Shamir secret sharing scheme [52] allows a dealer to partition a secret into n shares, t of which are required to recover the secret. Shamir secret sharing is information-theoretically secure.

Definition 2 (Shamir secret sharing [52]). *Shamir secret sharing* Shamir *is a threshold secret sharing scheme that consists of the following algorithms:*

- $\mathsf{Shamir.Share}(s, n, t) \to \{(1, \phi_1), \ldots, (n, \phi_n)\}$*: On input a secret* s*, the number of participants* n*, and a threshold* t*, perform the following. First, define a polynomial* $f(x) = s + a_1 + a_2 x^2 + \cdots + a_{t-1} x^{t-1}$ *by sampling* $t - 1$ *coefficients at random* $(a_1, \ldots, a_{t-1}) \xleftarrow{\$} \mathbb{Z}_q$*. Then, set each participant's share* $\phi_i, i \in [n]$*, to be the evaluation of* $f(i)$*:* $\phi_i \leftarrow s + \sum_{j \in [t-1]} a_j i^j$*. Output* $\{(i, \phi_i)\}_{i \in [n]}$*.*
- $\mathsf{Shamir.Recover}(t, \{(i, \phi_i)\}_{i \in \eta}) \to \perp/\mathsf{sk}$*: On input a threshold* t *and a set of shares* $\{(i, \phi_i)\}_{i \in \eta}$*, output* $\perp$ *if* $\eta \not\subseteq [n]$ *or if* $|\eta| < t$*. Otherwise, compute* $L(x) = \sum_{i \in \eta} w_i L_i(x) = \sum_{i \in \eta} w_i \prod_{j \in \eta, j \neq i} \frac{x - j}{i - j}$*. If* $\deg(L_i) > t - 1$*, return* $\perp$*. Otherwise, return* $s = L(0) = \sum_{i \in \eta} w_i L_i(0) = \sum_{i \in \eta} w_i \prod_{j \in \eta, j \neq i} \frac{j}{j - i}$*.*

3.3 General Forking Lemma

We next review the general forking lemma by Bellare and Neven [4], which itself is a formalization of the forking lemma introduced by Pointcheval and Stern [46]. We show the general forking algorithm in Figure 1.

Lemma 1 (General Forking Lemma [4]). *Let* H *be a finite set and* $r \geq 1$ *be an integer. Let* IG *be a randomized instance generator and let* $X \xleftarrow{\$} \mathsf{IG}$ *be an instance. Let* $\mathcal{C}$ *be a randomized algorithm that takes as input* X*, quantities* $h_1, \ldots, h_r \in H$*, and a random tape* ρ*. Let* $\mathsf{accept}(\mathcal{C})$ *be the probability that* $\mathcal{C}$ *outputs an accepting answer, namely*

$$\mathsf{accept}(\mathcal{C}) := \Pr\left[j \neq \perp \ : \ \begin{array}{c} X \xleftarrow{\$} \mathsf{IG}, \quad h_1, \ldots, h_r \xleftarrow{\$} H \\ (j, \mathsf{aux}) \xleftarrow{\$} \mathcal{C}\big(X, (h_1, \ldots, h_r); \rho\big) \end{array} \right].$$

Let $\mathsf{Fork}^{\mathcal{C}}(X)$ *be the general forking algorithm defined in Figure 1 and let*

$$\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) := \Pr\left[\alpha \neq \perp \ : \ X \xleftarrow{\$} \mathsf{IG}, \quad \alpha \xleftarrow{\$} \mathsf{Fork}^{\mathcal{C}}(X) \right].$$

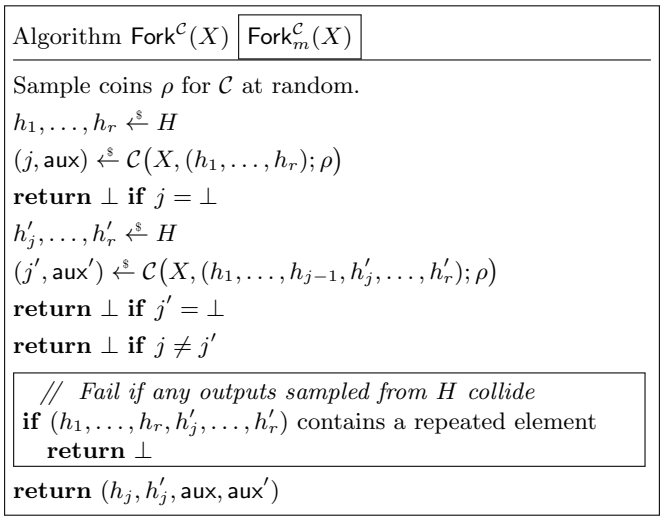


Fig. 1: The general forking algorithm $\mathsf{Fork}^{\mathcal{C}}(X)$ and the modified general forking algorithm $\mathsf{Fork}^{\mathcal{C}}_m(X)$, defined with respect to an algorithm $\mathcal{C}$ and instance X. The difference between the general forking algorithm and modified variant is shown in a box, for emphasis. In summary, the modified general forking algorithm aborts if *any* of the $(h_1, \ldots, h_r, h'_j, \ldots, h'_r)$ collide.

Then, $\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}})$ *is bounded by*

$$\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) \geq \mathsf{accept}(\mathcal{C}) \cdot \left(\frac{\mathsf{accept}(\mathcal{C})}{r} - \frac{1}{|H|} \right).$$

A modified forking lemma. We will employ a slight modification of the forking experiment, and give a corollary on how it impacts the accepting probability of its output.

The modification to the forking experiment is natural; intuitively, we add an additional abort condition if any of the values $h_1, \ldots, h_r, h'_j, \ldots, h'_r$ collide. Because there are at most $2r$ values, and they are all sampled uniformly at random from the set H, the probability that any of them collide is at most $2r^2/|H|$. By considering this case here, we can be sure that such collisions are considered in our proof of security for Arctic.

Corollary 1. *Let* $\mathsf{Fork}^{\mathcal{C}}_m$ *be the forking experiment in Figure 1. Then using the notation of Lemma 1 we have*

$$\begin{aligned} \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}_m) &\geq \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) - \Pr[\mathsf{BadHashEvent}] \\ &\geq \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) - 2r^2/|H| \end{aligned} \tag{1}$$

Where BadHashEvent *denotes the event that* $\mathsf{Fork}^{\mathcal{C}}_m$ *returns* $\perp$ *due to the boxed lines in Figure 1.*

3.4 Deterministic Threshold Signature Schemes

We begin with the definition of a deterministic threshold signature scheme, and then define the notion of unforgeability. We build upon standard definitions and notions of unforgeability for threshold signatures in the literature [17, 25, 27, 29].

Intuitively, our definition for deterministic threshold signature schemes is identical to that of randomized threshold signature schemes, with the exception that in the deterministic setting, the signing algorithms are deterministic and the signer does not maintain state between rounds. Furthermore, each signing round in the deterministic setting is given as input the message m, coalition of signers C, and secret signing key share sk_i.

Definition 3 (Deterministic Threshold Signatures). *A* **deterministic threshold signature scheme** DT *with an interactive signing protocol consisting of* r *rounds is a tuple of PPT algorithms* $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}_1, \ldots, \mathsf{Sign}_r), \mathsf{Combine}, \mathsf{Verify})$*, defined as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: Accepts as input a security parameter* λ *and outputs public parameters* par*, which are then implicitly given as input to all other algorithms.*
- $\mathsf{KeyGen}(n, t, \mu) \to (\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]})$*: A probabilistic algorithm that takes as input the total number of possible signers* n*, the corruption threshold* t*, and the minimum number of participating signers* μ*. Outputs the public key* pk *representing the set of* n *signers, the set* $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}$ *of public and secret key shares for each signer.*
- $(\mathsf{Sign}_1, \ldots, \mathsf{Sign}_r) \to \{\rho_1^{(k)}, \ldots \rho_r^{(k)}\}_{i\in C}$*: A set of deterministic algorithms where each algorithm represents a single stage in an interactive signing protocol performed by signing party* $k \in [n]$ *in a signing set* $C \subseteq [n], |C| \geq \mu$ *with respect to a message* m*, defined as follows:*

$$\begin{aligned}
\rho_1^{(k)} &\leftarrow \mathsf{Sign}_1(k, \mathsf{sk}_k, \mathsf{m}, C) \\
\rho_2^{(k)} &\leftarrow \mathsf{Sign}_2(k, \mathsf{sk}_k, \mathsf{m}, C, \{\rho_1^{(i)}\}_{i\in C}) \\
&\vdots \\
\rho_r^{(k)} &\leftarrow \mathsf{Sign}_r(k, \mathsf{sk}_k, \mathsf{m}, C, \{\rho_{r-1}^{(i)}\}_{i\in C})
\end{aligned}$$

  where $\rho_1^{(k)}, \ldots, \rho_r^{(k)}$ *are protocol messages produced by party* $k \in C$*.*
- $\mathsf{Combine}(\mathsf{pk}, \mathsf{m}, C, \{(\rho_1^{(i)}, \ldots, \rho_r^{(i)})\}_{i\in C}) \to \sigma$*: A deterministic algorithm that takes as input the public key* pk*, the message* m*, the set of signers* C*, and the set of protocol messages sent by each party during the* $\mathsf{Sign}_1, \ldots, \mathsf{Sign}_r$ *signing stages, and outputs a joint signature* σ*.*
- $\mathsf{Verify}(\mathsf{pk}, \mathsf{m}, \sigma) \to 0/1$*: A deterministic algorithm that takes as input the public key* pk*, a message* m*, and signature* σ *and outputs* 1 *to indicate accept if the signature verifies; otherwise, it outputs* 0 *to indicate reject.*

Remark 1 (Distributed key generation). Our definition assumes a centralized key generation algorithm KeyGen to generate the public key pk and public key shares $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}$. However, our scheme and proofs can be adapted to use a fully decentralized distributed key generation protocol (DKG).

Remark 2 (Deferring the Choice of Coalition to Later Rounds.). Our definition likewise assumes that the coalition of signers C is provided in the first round of signing Sign_1. However, some constructions (as is the case with Arctic) may defer the choice of C to later rounds.

Correctness. A deterministic threshold signature scheme DT is *correct* if for all security parameters λ, all allowable $1 \leq t \leq \mu \leq n$, all $C \subseteq [n]$ such that $\mu \leq |C| \leq n$, and for all messages $m \in \{0,1\}^*$, the following relation holds:

$$\begin{aligned}
&\mathsf{DT.Verify}(\mathsf{pk}, \mathsf{m}, \sigma) = 1, \text{ for} \\
(\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}) &\xleftarrow{\$} \mathsf{DT.KeyGen}(n, t, \mu), \text{ where} \\
&\rho_1^{(k)} \leftarrow \mathsf{DT.Sign}_1(k, \mathsf{sk}_k, \mathsf{m}, C) \\
&\vdots \\
\rho_r^{(k)} &\leftarrow \mathsf{DT.Sign}_r(k, \mathsf{sk}_k, \mathsf{m}, C, \{\rho_{r-1}^{(i)}\}_{i\in C}), \text{ and} \\
\sigma &\leftarrow \mathsf{DT.Combine}(\mathsf{pk}, \mathsf{m}, C, \{\rho_1^{(i)}, \ldots, \rho_r^{(i)}\}_{i\in C})
\end{aligned}$$

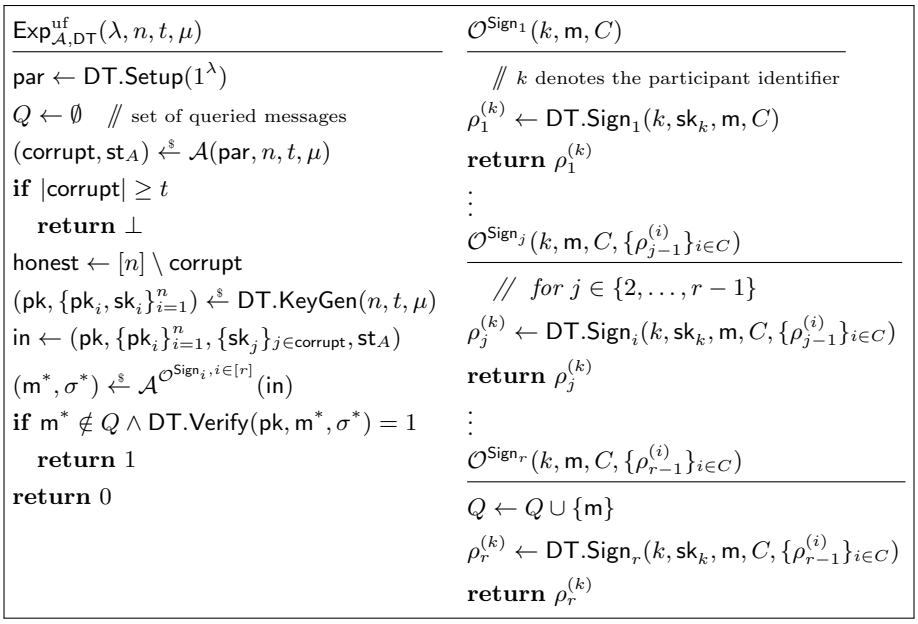


Fig. 2: Unforgeability game for a deterministic threshold signature scheme DT with a r-round signing protocol. The game assumes a static adversary that picks the players it corrupts at the beginning of the game. The public parameters par are implicitly given as input to all algorithms, and $\rho_1^{(k)}, \ldots, \rho_r^{(k)}$ represent protocol messages sent by participant k throughout the interactive signing protocol.

Unforgeability We present a game-based definition of unforgeability for a deterministic threshold signature scheme in Figure 2. This notion of unforgeability is analogous to the standard notion of chosen message attack (EUF-CMA) for standard signature schemes [30]. In this model, we present the adversary as a static adversary, which is allowed to corrupt at most $t-1$ signers. The static unforgeability game for a deterministic threshold signature scheme is nearly identical to that of a randomized scheme, with the exception that the environment does not maintain state between invocations of signing oracles by the adversary, and the adversary provides as input the message m and coalition of signers C (of its choosing) for each call to a signing oracle.

In more detail, in the unforgeability game, the environment generates public parameters par, and then allows the adversary to sample the corrupt parties $\mathsf{corrupt} \subset [n]$. If the set of corrupt parties is less than the corruption threshold t, it derives the set of honest parties $\mathsf{honest} \subseteq [n]$ as the remaining parties in $[n]$. The environment then runs KeyGen to derive the joint public key pk representing the set of n signers, the set of public key shares $\{\mathsf{pk}_i\}_{i \in [n]}$, and the secret key shares $\{\mathsf{sk}_i\}_{i \in [n]}$. The adversary is then run on input $n, t, \mu, \mathsf{par}, \mathsf{pk}, \{\mathsf{pk}_i\}_{i \in [n]}$, and the corrupt signing key shares $\{\mathsf{sk}_j\}_{j \in \mathsf{corrupt}}$. The adversary can then query any honest signers $k \in \mathsf{honest}$ of its choosing at each step in the signing protocol, by querying oracles $\mathcal{O}^{\mathsf{Sign}_1}, \ldots, \mathcal{O}^{\mathsf{Sign}_r}$, and has full power over choosing the set of signers C and the message m. Additionally, the adversary may query the signing round oracles in arbitrary order. However, unlike in the randomized setting, the environment for a deterministic threshold signature scheme does not maintain any session identifiers, or state about ongoing signing sessions. The adversary wins if it can produce a valid forgery σ^* with respect to the joint public key pk on a message m^* that has not been queried to an honest signer that has not been queried to $\mathcal{O}^{\mathsf{Sign}_r}$ (i.e., in the final round of signing). Importantly, this definition allows the adversary to be *rushing*, meaning it can wait to produce its outputs after having seen the honest outputs first. The adversary may also be *concurrent*, meaning it can open simultaneous signing sessions at once, or choose not to complete a signing session.

Definition 4 (Unforgeability). *Let the advantage of a static adversary $\mathcal{A}$ playing the unforgeability game against a deterministic threshold signature scheme DT as defined in Figure 2 be as follows:*

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{A},\mathsf{DT}}(\lambda, n, t, \mu) = \left|\Pr[\mathit{Exp}^{uf}_{\mathcal{A},\mathsf{DT}}(\lambda, n, t, \mu) = 1]\right|$$

A deterministic threshold signature scheme DT *is* unforgeable *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{A},\mathsf{DT}}(\lambda, n, t, \mu)$ *is negligible in* λ, *for all allowable* $n, t, \mu \in \mathbb{N}$.

4 Verifiable Pseudorandom Secret Sharing

We now introduce an extension to pseudorandom secret sharing that we call a *Verifiable Pseodurandom Secret Sharing* (VPSS) scheme. We begin by motivating the need for a VPSS, and then formally define VPSS and its security properties. Finally, in Section 4.3, we give a concrete VPSS construction, Shine, that we later use as a building block for Arctic.

4.1 Motivation

A verifiable random function (VRF) [41] is a keyed PRF, such that the PRF can be evaluated using only knowledge of a secret key, but is publicly verifiable given a public key. In particular, in addition to outputting the evaluation of the VRF, it also outputs a zero-knowledge proof that the VRF was evaluated correctly. A *distributed* VRF [19, 23, 42] allows the evaluation algorithm to be partitioned among a set of participants, all whom are equally trusted.

However, the use case of generating deterministic nonces for threshold Schnorr signatures presents a slightly different challenge. Instead of directly verifying that the nonce r_i was correctly generated, we instead need to verify correctness in zero-knowledge, with respect to a commitment $R_i = g^{r_i}$. One approach in the literature is to employ non-algebraic pseudorandom functions to generate the nonce, and then prove in zero-knowledge the correctness of the corresponding commitment [24, 44]. Unfortunately, generating such zero-knowledge proofs requires higher computational and complexity overhead.

We instead take a different approach towards verifying that a distributed pseudorandom function was correctly performed by a set of parties. Instead of each participant outputting a zero-knowledge proof that they individually performed the action correctly, we observe that the correctness of the evaluation can instead be *collectively* publicly verified, assuming some threshold of honest participants. Such an observation leads naturally to employing a pseudorandom secret sharing scheme [16], which is categorized by all parties holding a set of secret key shares, and individually and non-interactively being able to generate secret shares of additional pseudorandom values. We show that pseudorandom secret sharing schemes has a natural extension to the publicly verifiability setting. Finally, we give a concrete and efficient VPSS scheme, where all participants can publish commitments to their generated shares, and perform polynomial interpolation over elements in the public domain to ensure correctness.

We next build upon these observations by formalizing the notion of a verifiable pseudorandom secret sharing scheme.

4.2 VPSS Definition and Notions of Security

We now present an extension on pseudorandom secret sharing that we call a *Verifiable Pseudorandom Secret Sharing* (VPSS) scheme. In particular, a VPSS defines an additional verify algorithm to ensure that the pseudorandom function was performed correctly by collectively verifying outputs by all participants.

We now more formally define a VPSS, and its required security properties.

Definition 5. *A* **Verifiable Pseudorandom Secret Sharing** *scheme is the tuple of algorithms* $\mathsf{VPSS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Gen}, \mathsf{Verify}, \mathsf{Agg}, \mathsf{Recover})$, *such that:*

- $\mathsf{Setup}(1^\lambda)$*: Accepts as input the security parameter* λ, *and outputs public parameters* par, *where* par *is given as implicit input to all other algorithms.*

– $\mathsf{KeyGen}(n, t, \mu) \to \perp/(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*: A probabilistic algorithm that accepts as input the total number of participants* n*, a corruption threshold* t*, and the minimum number of parties* μ *required to participate in* Gen*. On failure, output* $\perp$*, otherwise, output* n *secret keys, one for each of the* n *participants.*

– $\mathsf{Gen}(k, \mathsf{sk}_k, w) \to (d_k, D_k)$*: A deterministic algorithm that accepts as input a participant identifier* k*, the secret key for that participant* sk_k*, and some input* w*. Generates the pseudorandom secret share* $d_k \in \mathcal{P}$ *from some domain* $\mathcal{P}$*, using* sk_k *as the random seed and* w *as the corresponding input. Then, generates its public commitment* $D_k \in \mathcal{O}$ *to* d_k *from some domain* $\mathcal{O}$*. Outputs* (d_k, D_k)*.*

– $\mathsf{Verify}(t, \mu, C, \{D_j\}_{j \in C}) \to \{0, 1\}$*: A deterministic algorithm that accepts as input the corruption threshold* t*, the minimum number of participants* μ*, a coalition of participants* C *such that* $|C| \geq \mu$*, and a set of commitments to pseudorandom secret shares* $(D_j)_{j \in C}$ *for that coalition. Outputs* 1 *to indicate that the secret sharing was correctly performed, otherwise, output* 0*.*

– $\mathsf{Agg}(t, \mu, C, \{D_j\}_{j \in C}) \to D$*: A deterministic algorithm that accepts as input the corruption threshold* t*, the minimum number of participants* μ*, a coalition of participants* C *such that* $|C| \geq \mu$*, and the set of commitments to pseudorandom secret shares. Outputs the commitment to the aggregated pseudorandom secret* D*.*

– $\mathsf{Recover}(t, \mu, C, \{d_j\}_{j \in C}) \to \perp/(d, D)$*: A deterministic algorithm that accepts a corruption threshold* t*, the minimum number of participants* μ*, a coalition* $C \subseteq [n], |C| \geq \mu$*, and a set of shares* $\{d_j\}_{j \in C}$*. It outputs* $\perp$ *if* $C \not\subseteq [n]$*,* $|C| < \mu$*, or if the shares are inconsistent. Otherwise, it recovers* d *using the set of shares, derives the corresponding commitment* D*, and outputs* (d, D)*.*

Correctness. A VPSS is **correct** if for all λ, possible inputs w and choices of $t, \mu, n \in \mathbb{N}$ where $t \leq \mu \leq n$, when $\mathsf{par} \leftarrow \mathsf{VPSS.Setup}(1^\lambda)$ and $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VPSS.KeyGen}(n, t, \mu)$, there exists $d \in \mathcal{P}, D \in \mathcal{O}$ such that Equation 2 holds:

$$\begin{aligned} &\text{for all } i \in [n] \text{ and for all } C_\ell \subseteq [n], |C_\ell| \geq \mu, \text{ when} \\ &\mathsf{VPSS.Gen}(i, \mathsf{sk}_i, w) \to (d_i, D_i), \text{ then} \\ &\mathsf{VPSS.Verify}(t, \mu, C_\ell, \{D_j\}_{j \in C}) = 1, \text{ and} \\ &\mathsf{VPSS.Recover}(t, \mu, C_\ell, (d_j)_{j \in C}) \to (d, D), \text{ and} \\ &\mathsf{VPSS.Agg}(t, \mu, C_\ell, (D_j)_{j \in C}) \to D \end{aligned} \tag{2}$$

Security. Similarly to verifiable random functions [19,23,41,42], we say that a VPSS is *secure* if it is unique, verifiable, and pseudorandom. We define these notions next.

Uniqueness. Intuitively, a VPSS is unique if it produces exactly one commitment to a (pseudorandom) value for each input w, regardless of the choice of coalition. More specifically, picking different coalitions should always result in the same commitment D, when the input w remains the same. We show the VPSS uniqueness experiment in Figure 3.

In the uniqueness experiment, the adversary is allowed to query honest participants for shares and commitments on inputs of its choosing. The adversary then outputs the evaluation input w, two coalitions C, C', and two sets of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}}$, $(D'_i)_{i \in C' \cap \mathsf{corrupt}}$.

The environment then derives the honest players' shares and commitments on w, producing $(d_j, D_j)_{j \in \mathsf{honest}}$. After deriving the sets $S_1 \leftarrow (D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D_i)_{i \in C \cap \mathsf{honest}}$ and $S_2 \leftarrow (D'_i)_{i \in C' \cap \mathsf{corrupt}} \cup (D_i)_{i \in C' \cap \mathsf{honest}}$, the adversary loses if the verification algorithm outputs 0 on either set. If both sets verify, the adversary wins if the corresponding commitments for each set are not equal.

We define uniqueness for a VPSS more formally in Definition 6.

Definition 6. *Let the advantage of an adversary* $\mathcal{A}$ *playing the uniqueness game as defined in Figure 3 be as follows:*

$\mathsf{Exp}^{\mathrm{uq}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu)$

$\mathsf{par} \leftarrow \mathsf{VPSS.Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
return $\perp$ **if** $|\mathsf{corrupt}| \geq t$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VPSS.KeyGen}(n, t, \mu)$
$(w, \mathsf{out}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
$(C, (D_i)_{i \in C \cap \mathsf{corrupt}}), (C', (D'_i)_{i \in C' \cap \mathsf{corrupt}}) \leftarrow \mathsf{out}$
$(d_j, D_j) \leftarrow \mathsf{VPSS.Gen}(j, \mathsf{sk}_j, w), j \in \mathsf{honest}$
$S_1 \leftarrow (D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D_i)_{i \in C \cap \mathsf{honest}}$
$S_2 \leftarrow (D'_i)_{i \in C' \cap \mathsf{corrupt}} \cup (D_i)_{i \in C' \cap \mathsf{honest}}$
return 0 **if** $\mathsf{VPSS.Verify}(t, \mu, C, S_1) \neq 1$
return 0 **if** $\mathsf{VPSS.Verify}(t, \mu, C', S_2) \neq 1$
$D \leftarrow \mathsf{VPSS.Agg}(t, \mu, C, S_1)$
$D' \leftarrow \mathsf{VPSS.Agg}(t, \mu, C', S_2)$
if $D \neq D'$
  return 1
return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w_i)$

// k denotes the participant identifier
$(d_k, D_k) \leftarrow \mathsf{VPSS.Gen}(k, \mathsf{sk}_k, w_i)$
return (d_k, D_k)

Fig. 3: Uniqueness game for a VPSS.

$$\mathsf{Adv}^{\mathrm{unq}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = \left|\Pr[\mathit{Exp}^{uq}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = 1]\right|$$

A VPSS VPSS *is* **unique** *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{unq}}_{\mathcal{A},\mathsf{VPSS}}$ *is a negligible function of* λ, *for* $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.

Verifiability. Intuitively, a VPSS is verifiable if given a set of commitments to shares from a coalition of participants, the verify algorithm will detect if some subset of players deviated from the protocol. We show the VPSS verifiability experiment in Figure 4.

In the experiment, the adversary is allowed to query honest participants for shares and commitments on inputs of its choosing. The adversary then outputs a coalition C, a VPSS input w, and a set of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}}$. The environment then follows the protocol, deriving both the corrupt and honest players' commitments on w, and producing $(D'_j)_{j \in C}$. The adversary loses if its output is identical to the honestly derived commitments for the corrupted players. Then, the environment checks if the set of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D'_j)_{j \in C \cap \mathsf{honest}}$ are valid with respect to C and w. If so, the adversary wins, otherwise, it loses.

We define verifiability for a VPSS more formally in Definition 7.

Definition 7. *Let the advantage of an adversary* $\mathcal{A}$ *playing the verifiability game as defined in Figure 4 be as follows:*

$$\mathsf{Adv}^{\mathrm{verf}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = \left|\Pr[\mathit{Exp}^{verf}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = 1]\right|$$

A VPSS VPSS *is* **verifiable** *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{verf}}_{\mathcal{A},\mathsf{VPSS}}$ *is a negligible function of* λ, *for* $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.

$\mathsf{Exp}^{\text{verf}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu)$

$\mathsf{par} \leftarrow \mathsf{VPSS.Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
return $\perp$ **if** $|\mathsf{corrupt}| \geq t$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VPSS.KeyGen}(n, t, \mu)$
$(w, C, (D_j)_{j \in C \cap \mathsf{corrupt}}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
for $j \in C$ **do**
  $(d'_j, D'_j) \leftarrow \mathsf{VPSS.Gen}(j, \mathsf{sk}_j, w)$
if $(D_j)_{j \in C \cap \mathsf{corrupt}} = (D'_j)_{j \in C \cap \mathsf{corrupt}}$
  return 0
 // *$\mathcal{A}$ must deviate from the protocol to win*
$S \leftarrow (D_k)_{k \in C \cap \mathsf{corrupt}} \cup (D'_j)_{j \in C \cap \mathsf{honest}}$
if $\mathsf{VPSS.Verify}(t, \mu, C, S) = 1$
  return 1
return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w_i)$

// k denotes the participant id
$(d_k, D_k) \leftarrow \mathsf{VPSS.Gen}(k, \mathsf{sk}_k, w_i)$
return (d_k, D_k)

Fig. 4: Verifiability game for a VPSS.

Pseudorandom. Intuitively, a VPSS is pseudorandom if the adversary has negligible advantage distinguishing between a real VPSS output from one that is randomly sampled. We show the VPSS pseudorandomness experiment in Figure 5.

In the pseudorandomness experiment, the environment begins by picking a random bit $b \xleftarrow{\$} \{0, 1\}$. It then performs key generation, and initializes the adversary with the public parameters and the corrupted parties secret key shares.

The adversary is allowed to query $\mathcal{O}^{\mathsf{Gen}}$ on honest participants for shares and commitments on inputs of the adversary's choosing. If $b = 0$, the environment responds by following the VPSS protocol. If $b = 1$, the environment first checks to see if it has responded to the query before, replying with the response if so. Otherwise, it uses a simulating algorithm SimGen to simulate generating honest players' shares and commitments. The details of SimGen depend on the specifics of the VPSS construction.

The adversary outputs $(b', w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$, where b' is the adversary's guess for the value of b, and $(w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$ corresponds to one of the sessions the adversary initiated with $\mathcal{O}^{\mathsf{Gen}}$.

If $b = 1$, the environment checks if $(w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$ corresponds to an existing session and is consistent with the honest parties' commitments, outputting a random coin if it does not. Otherwise, the environment next checks that the aggregated commitment $\mathsf{VPSS.Agg}(t, \mu, C^*, \{D_k\}_{k \in C^*})$ is equal to the simulated commitment D for that session, if the check does not hold, the environment outputs 1.

Otherwise, the environment checks if the adversary's guess b' is equal to b; outputting 1 if the check holds, otherwise, outputting 0.

We define pseudorandomness for a VPSS more formally in Definition 8.

Definition 8. *Let the advantage of an adversary $\mathcal{A}$ playing the pseudorandomness game as defined in Figure 5 be as follows:*

$$\mathsf{Adv}^{\text{psdr}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = \left|\Pr[\mathit{Exp}^{\mathit{psdr}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu) = 1] - 1/2\right|$$

A VPSS VPSS *is* **pseudorandom** *if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\text{psdr}}_{\mathcal{A},\mathsf{VPSS}}$ is a negligible function of λ, for $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.*

$\mathsf{Exp}^{\mathrm{psdr}}_{\mathcal{A},\mathsf{VPSS}}(\lambda, n, t, \mu)$

$b \xleftarrow{\$} \{0,1\}$

$z \xleftarrow{\$} \{0,1\}$ // *random coin for trivial losing conditions*

$Q \leftarrow \emptyset$

$\mathsf{par} \leftarrow \mathsf{VPSS.Setup}(1^\lambda)$

$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$

return $\perp$ **if** $|\mathsf{corrupt}| \geq t$

$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$

$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VPSS.KeyGen}(n, t, \mu)$

$(b', w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$

if $b = 1$

  return z **if** $Q[w^*] = \perp$

  $(D, \{D_i\}_{i \in \mathsf{honest}}) \leftarrow Q[w^*]$

  if $\mathsf{VPSS.Verify}(t, \mu, C^*, \{D_k\}_{k \in C^*}) \neq 1$

    return z

  if $\mathsf{VPSS.Agg}(t, \mu, C^*, \{D_k\}_{k \in C^*}) \neq D$

    return 1

return 1 **if** $b \stackrel{?}{=} b'$

return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w)$

// k denotes the participant identifier

return $\perp$ **if** $k \notin \mathsf{honest}$

if $b = 0$

  $(d_k, D_k) \leftarrow \mathsf{VPSS.Gen}(k, \mathsf{sk}_k, w)$

  return (d_k, D_k)

// $b = 1$ *case*

if $Q[w] \neq \perp$

  $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow Q[w]$

else

  $d' \xleftarrow{\$} \mathcal{P}$

  $\mathsf{in} \leftarrow (w, d', \mathsf{corrupt}, n, (\mathsf{sk}_j)_{j \in \mathsf{corrupt}})$

  $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(\mathsf{in})$

  $Q[w] = (D, \{(d_i, D_i)\}_{i \in \mathsf{honest}})$

  // SimGen *is a simulating algorithm*

  // *defined by the VPSS construction.*

return (d_k, D_k)

Fig. 5: Pseudorandomness game for a VPSS.

4.3 Shine, A Concrete Verifiable Pseudorandom Secret Sharing Scheme

We now define a concrete VPSS that we call Shine, that builds upon the pseudorandom secret sharing scheme as defined by Cramer Damgård, and Ishai [16]. However, Shine additionally defines a verify algorithm that is publicly verifiable (assuming the authenticity of the inputs), as well as an algorithm to combine participant commitments.

As a reminder, the pseudorandom secret sharing scheme by Cramer, Damgård, and Ishai [16] itself builds on replicated secret sharing [32]. However, Cramer et al. define a mechanism to non-interactively convert shares of a replicated secret sharing scheme to shares of a Shamir-secret-shared value. They then show how this mechanism can be used as a distributed pseudorandom function; i.e., given a replicated secret sharing of a random value, participants can generate Shamir secret shares of an unbounded number of pseudorandom values, without interaction.

We extend this pseudorandom secret sharing scheme by Cramer et al. and additionally define a public verifiability mechanism. More specifically, we define a Verify function that uses the set of participants' commitments to ensure all participants performed the evaluation step in a consistent manner. Such consistency checks have been used in prior literature [6], but our check is performed over commitments to secret shares, as opposed to verifying the shares directly. We prove that Shine is secure assuming a cryptographically secure hash function, when at minimum $2t - 1$ parties participate in evaluation and up to $t - 1$ participants are corrupted (honest majority).

We now define Shine in more detail. We show KeyGen for Shine as a centralized operation, but this operation can easily be decentralized. Shine is additionally defined with respect to a hash function H, where $\mathsf{H} : \mathbb{Z}_q \times \{0,1\}^* \to \mathbb{Z}_q$ is a cryptographically secure hash function.

- $\mathsf{Shine.Setup}(1^\lambda)$: Accepts as input the security parameter λ, outputs public parameters $\mathsf{par} = (\mathbb{G}, q, g)$, where $(\mathbb{G}, q, g)$ is generated by GroupGen. par is given as an implicit input to all other algorithms.

– $\mathsf{Shine[H].KeyGen}(n, t, \mu) \to \perp/(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$: On input the total number of participants n, the corruption threshold t, and minimum number of participants μ, the dealer performs replicated secret sharing of a random secret $\mathsf{sk} \in \mathbb{Z}_q$ [16], following the following steps:
  1. First, the dealer checks if $\mu \geq 2t - 1$; if the check fails, output $\perp$.
  2. Let A be the set such that $A = \binom{[n]}{t-1}$, and let γ be the size of A; i.e, $\gamma = |A| = \binom{n}{t-1}$.
  3. Generate γ secret shares $\{\phi_i\}_{i\in[\gamma]}$, by sampling $\phi_i \xleftarrow{\$} \mathbb{Z}_q, i \in [\gamma]$. Implicitly, the secret sk is such that $\mathsf{sk} = \sum_{i\in\gamma} \phi_i$.
  4. Initialize empty sets $\mathsf{sk}_1 = \emptyset, \ldots, \mathsf{sk}_n = \emptyset$.
  5. For each set $a_i \in A, i \in [\gamma]$ and each participant identifier $j \in [n] \setminus a_i$, append $\mathsf{sk}_j \leftarrow \mathsf{sk}_j \cup \{(a_i, \phi_i)\}$.
  6. Output $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$. Each sk_j is a set of size $\delta = \binom{n-1}{t-1}$, consisting of the tuples (a_i, ϕ_i). Each a_i is itself a set, such that $a_i \subset [n]$ and $|a_i| = t - 1$, where if $a_i \in \mathsf{sk}_j$, then $j \notin a_i$.
– $\mathsf{Shine[H].Gen}(k, \mathsf{sk}_k, w) \to (d_k, D_k)$: On input participant identifier $k \in [n]$, secret key share sk_k, and input $w \in \{0,1\}^*$, perform the following steps:
  1. Parse $\{(a_i, \phi_i)\}_{i\in[\delta]} \leftarrow \mathsf{sk}_k$.
  2. For each set a_i, let $L'_{a_i}(x)$ be the degree $t-1$ polynomial defined by the set a_i, as given in Equation 3:
$$L'_{a_i}(x) = \prod_{j\in a_i} \frac{j - x}{j} \tag{3}$$
  Note that $L'_{a_i}(j) = 0$ for all $j \in a_i$ and $L'_{a_i}(0) = 1$.
  3. Obtain the share via Equation 4:
$$d_k \leftarrow \sum_{i\in[\delta]} \mathsf{H}(\phi_i, w) \cdot L'_{a_i}(k) \tag{4}$$
  Note that the $L'_{a_i}(k)$ values can be precomputed by participant k, as they are independent of w, so this step requires δ evaluations of H and δ multiplications and additions in $\mathbb{Z}_q$.
  4. Derive the commitment to d_k as $D_k \leftarrow g^{d_k}$.
  5. Output (d_k, D_k).
– $\mathsf{Shine[H].Verify}(t, \mu, C, \{D_j\}_{j\in C}) \to \{0, 1\}$: Perform the following steps:
  1. If $|C| \not\geq \mu$, return 0.
  2. Otherwise, define $B = (B_0, B_1, \ldots, B_{|C|-1})$ to be the tuple of $|C|$ commitments to *coefficients* of a polynomial f defined "in the exponent," where each $D_i, i \in C$ is a commitment to a *point* on the same polynomial f. Each B_i can be derived via Equation 5:
$$B_i = \prod_{j\in C} D_j^{L_{j,i}} \tag{5}$$
  where $L_{j,i}$ is the coefficient of the i^{th} term x^i of the j^{th} Lagrange polynomial $L_j(x)$ for the coalition C, as described in Section 3.1.
  3. Let $\ell = |C| - t$. The verifier now checks that all participants followed the protocol honestly, by checking in the exponent that their shares lie on a polynomial of degree $t - 1$. In particular, the verifier ensures that B is a commitment to a polynomial of degree at most $t - 1$,[1] by checking the last ℓ entries in B are equal to the identity of $\mathbb{G}$:[2]
$$B_{t-1+i} = I_{\mathbb{G}}, \forall\ i \in [\ell] \tag{6}$$

[1] When at least t parties follow the protocol honestly, their shares will completely define this polynomial.
[2] As an optimization, note that $B_1, \ldots, B_{t-1}$ need not be computed.

4. If the check in Equation 6 holds, output 1, otherwise, output 0.

– $\mathsf{Shine}[\mathsf{H}].\mathsf{Agg}(t, \mu, C, \{D_j\}_{j\in C}) \to D$: Accepts as input the corruption threshold t, minimum participants μ, the coalition C, and set of (verified) commitments $\{D_j\}_{j\in C}$. The commitments can be combined as in Equation 7, where the coefficients λ_j are determined by the coalition C.

$$D \leftarrow \prod_{j\in C} D_j^{\lambda_j} = B_0 \tag{7}$$

Output D.

– $\mathsf{Shine}[\mathsf{H}].\mathsf{Recover}(t, \mu, C, \{d_j\}_{j\in C}) \to \perp/(d, D)$: Receives as input the corruption threshold t, the minimum number of participants μ, the coalition C, and the set of pseudorandom secret shares $\{d_j\}_{j\in C}$. Performs the following steps:

1. If $|C| < \mu$, or $C \not\subset [n]$, output $\perp$.
2. For each $d_j, j \in C$, derive $D_j \leftarrow g^{D_j}$.
3. If $\mathsf{Shine.Verify}(t, \mu, C, \{D_j\}_{j\in C}) \neq 1$, then output $\perp$.
4. Otherwise, the shares can be combined as in Equation 8, where the λ_j are determined by the coalition C.
$$d \leftarrow \sum_{j\in C} d_j \cdot \lambda_j \tag{8}$$
5. Derive the commitment to d as $D \leftarrow g^d$.
6. Output (d, D).

Correctness. By correctness of pseudorandom secret sharing as defined by Cramer et al. [16], when key generation is honestly performed, the Gen algorithm produces secret shares $d_i = f(i)$ that are points on a degree $t-1$ Shamir secret sharing polynomial $f(x) = \sum_{i\in[\gamma]} \mathsf{H}(\phi_i, w) L'_{a_i}(x)$. The combined commitment $d = \sum_{i\in C} d_i \lambda_i$ for $C \subseteq [n], |C| \geq 2t-1$ is a commitment to a pseudorandom secret value $d = f(0) = \sum_{i\in[\gamma]} \mathsf{H}(\phi_i, w)$. Agg and $\mathsf{Recover}$ simply perform polynomial interpolation of these values with respect to the set of participants, and so are likewise correct.

Verify is correct because honest parties output commitments $D_i = g^{f(i)}$ for the degree $t-1$ polynomial f defined above. Therefore, if all parties are honest, the coefficients of $x^t, \ldots, x^{|C|-1}$ of $f(x)$ will be 0, so the commitments $B_t, \ldots, B_{|C|-1}$ to those coefficients will be $I_{\mathbb{G}}$ (the identity element of $\mathbb{G}$), and so Verify will output 1.

Security Shine is verifiable, unique, and pseudorandom. We give the corresponding proofs in Appendix B.

Theorem 1. *Shine is information-theoretically verifiable, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, and participants exchange messages over an authenticated channel.*

Theorem 2. *Shine is information-theoretically unique, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, and participants exchange messages over an authenticated channel.*

Theorem 3. *Shine is pseudorandom in the Random Oracle Model, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, Shine is verifiable, and where $(n, t) \in \mathbb{N}$ are such that $\binom{n-1}{t-1} = poly(n)$. Concretely,*

$$\mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{A},\mathsf{Shine}}(\lambda, n, t, \mu) \leq q_h/q \tag{9}$$

where q_h is the number of times $\mathcal{A}$ is allowed to query H.

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$
$\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3)$
return par
// par is given implicitly
// to all other algorithms

$\mathsf{KeyGen}(n, t, \mu)$

// Performed by a trusted
// dealer, or DKG
if $\mu < 2t - 1$ or $\mu > n$
  return $\perp$
// Require $\geq 2t - 1$ *signers*
$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$; $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$
$\{(i, \mathsf{sk}_i^{(2)})\}_{i=1}^n \xleftarrow{\$} \mathsf{Shamir.Share}(\mathsf{sk}, n, t)$
$(\mathsf{sk}_i^{(1)})_{i=1}^n \xleftarrow{\$} \mathsf{Shine}[\mathsf{H}_1].\mathsf{KeyGen}(n, t, \mu)$
for $i \in \{1, \ldots, n\}$ **do**
  $\mathsf{pk}_i^{(2)} \leftarrow g^{\mathsf{sk}_i^{(2)}}$
  $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$
  $\mathsf{pk}_i \leftarrow (\mathsf{pk}_i^{(2)})$
return $(\mathsf{pk}, \{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i \in [n]})$

$\mathsf{Sign}_1(k, \mathsf{sk}_k, \mathsf{m})$

$(\mathsf{sk}_k^{(1)}, \mathsf{sk}_k^{(2)}, \mathsf{pk}) \leftarrow \mathsf{sk}_k$
$y_k \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$
$(r_k, R_k) \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Gen}(k, \mathsf{sk}_k^{(1)}, y_k)$
return (y_k, R_k)

$\mathsf{Sign}_2(k, \mathsf{sk}_k, \mathsf{m}, C, \{(y_j, R_j)\}_{j \in C})$

return $\perp$ **if** $|C| < 2t - 1$
$(\mathsf{sk}_k^{(1)}, \mathsf{sk}_k^{(2)}, \mathsf{pk}) \leftarrow \mathsf{sk}_k$
$y' \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$
for $j \in C$ **do**
  if $y_j \neq y'$
    return $\perp$
(r_k, R'_k)
    $\leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Gen}(k, \mathsf{sk}_k^{(1)}, y')$
// Re-derive state from Sign_1
if $k \notin C$ or $R'_k \notin \{R_j\}_{j \in C}$
  return $\perp$
$\mathsf{input} \leftarrow (t, \mu, C, \{R_j\}_{j \in C})$
if $\mathsf{Shine}[\mathsf{H}_1].\mathsf{Verify}(\mathsf{input}) \neq 1$
  return $\perp$
$R \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Agg}(\mathsf{input})$
$c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, \mathsf{m})$
$z_k \leftarrow r_k + c \cdot \mathsf{sk}_k^{(2)}$
return z_k

$\mathsf{Combine}(\mathsf{pk}, \mathsf{m}, C, \{(y_j, R_j), z_j\}_{j \in C}))$

$\mathsf{input} \leftarrow (t, \mu, C, \{R_j\}_{j \in C})$
$R \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Agg}(\mathsf{input})$
$c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, \mathsf{m})$
// λ_j *are Lagrange*
// coefficients for C
$z \leftarrow \sum_{j \in C} z_j \lambda_j$
if $g^z \neq R \cdot \mathsf{pk}^c$
  return $\perp$
return $\sigma = (R, z)$

Fig. 6: Arctic, a deterministic threshold Schnorr signature scheme. For security, Arctic requires that the minimum number of signing parties $\mu \leq n$ be at minimum $\mu \geq 2t - 1$, where t is the tolerated corruption threshold. We further require that messages exchanged between participants are sent over an authenticated channel. Arctic builds upon the verifiable pseudorandom secret sharing scheme Shine defined in Section 4.3, as well as Shamir's secret sharing. Verification of signatures is identical to the single-party Schnorr verification algorithm.

5 Arctic, A Deterministic and Stateless Two-Round Threshold Schnorr Signature Scheme

We now introduce Arctic, an efficient, two-round, deterministic threshold Schnorr signature scheme for moderately sized groups of participants that does not require participants to keep state between rounds of the signing protocol. As a building block, Arctic uses Shine to generate nonces deterministically, and to verify that all other participants followed the protocol honestly. Arctic is secure assuming fewer than t participants are corrupted, and at least μ participated in the signing protocol, where $\mu \leq n$ but $\mu \geq 2t - 1$.

Remark 3 (Distributed Key Generation). The Arctic construction given in Figure 6 assumes a centralized key generation procedure; however, using a distributed key generation (DKG) scheme is equally possible.

Remark 4 (Requirement of Authenticated Channels). We require that the messages exchanged between participants in the Arctic construction shown in Figure 6 be sent over authenticated channels; i.e., messages must be authenticated and verifiable as having come from their purported senders. Otherwise, an adversary can simply pick contributions that are consistent with a single honest party's R_i, which would result in a valid input for $\mathsf{Shine.Verify}$. Note that we do *not* assume the authenticated channel maintains any session identifiers or state about messages; Arctic remains secure even if the adversary were to replay old authenticated messages. However, if a participant receives an unauthenticated message, we require that the participant aborts.

5.1 The Construction.

We now give more detail for each stage in Arctic; see Figure 6 for a high-level overview.

Key Generation. All participants with identifiers $i \in [n]$ begin by receiving a secret signing share $\mathsf{sk}_i^{(2)}$ and a public signing share $\mathsf{pk}_i^{(2)} = g^{\mathsf{sk}_i^{(2)}}$. In Figure 6, we show key generation as a centralized procedure, but a DKG can likewise be used. Each $\mathsf{sk}_i^{(2)}$ is a t-of-n Shamir secret sharing of the group's joint secret key sk; participants use these keys for signing messages. Participants also receive the public signing keys $\{\mathsf{pk}_i^{(2)}\}_{i\in[n]}$ for all other participants.

In addition, each participant receives a secret Shine key $\mathsf{sk}_i^{(1)}$ generated by performing $\mathsf{Shine.KeyGen}$. Participants use these keys to generate nonces and commitments for each signing session.

Each participant's public key share pk_i is just one public key, where $\mathsf{pk}_i = \mathsf{pk}_i^{(2)}$. Each participant's secret key share is the tuple $\mathsf{sk}_i = (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$.

Signing. In the first round of signing, each participant k receive as input a message m. First, each party derives $y_k \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$. To generate their nonce r_k and commitment R_k, each participant performs $(r_k, R_k) \xleftarrow{\$} \mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y_k)$. Each participant then outputs (y_k, R_k); they do not need to keep any state.

In the second round of signing, all participants again receive as input a message m, as well as a set C representing the indices of at least μ signers, where $C \subseteq [n], |C| \geq \mu$. Additionally, participants receive the list of tuples $\{(y_j, R_j)\}_{j\in C}$. First, each party re-derives $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$. Then, each party checks the consistency of all other parties' views of m by checking that for each $i \in C$, $y_i = y'$. Because we require that each protocol message is authenticated by its respective party, then this check guarantees that an adversarial player cannot split the view of honest players by sending different messages or choices of coalitions. If any check fails, the party aborts.

Otherwise, if all consistency checks succeed, each participant re-derives their nonce and commitment using Shine, again performing $(r_k, R'_k) \xleftarrow{\$} \mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y')$. Then, the participant checks its identifier is in the coalition C and that R'_k is in the set of commitments $(R_j)_{j\in C}$, aborting if either check does not hold. Then, each participant verifies that all other participants followed the protocol to derive their commitment, by checking $\mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j\in C})$. If the check fails, they abort the protocol.

Finally, if all of the above checks pass, each participant $k \in C$ will then derive the group commitment $R \leftarrow \mathsf{Shine.Agg}(t, \mu, C, \{R_j\}_{j\in C})$, and the challenge $c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, \mathsf{m})$. Finally, each participant derives their signature share $z_k \leftarrow r_k + c \cdot \mathsf{sk}_k^{(2)}$. Each participant outputs z_k as its output for Sign_2.

Combination and Verification. To perform the $\mathsf{Combine}$ algorithm, the group commitment R is first derived using values output by participants from Sign_1. Then, the response z is derived by finding $z \leftarrow \sum_{j \in C} z_j \lambda_j$, where the λ_j are the Lagrange coefficients for the set C. The output from $\mathsf{Combine}$ is the Schnorr signature $\sigma = (R, z)$, which can be verified using the single-party Schnorr verification algorithm given in Definition 1.

5.2 Security

Correctness. In the first round of signing, participants will output nonces and commitments $(r_i, R_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y_i)$, for $i \in C$, where $y_i \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$, and $R_i = g^{r_i}$.

In the second round of signing, each participant receives the set of tuples $\{(y_j, R_j)\}_{j \in C}$. Because H_2 is correct, then after deriving $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$, then the check that $y_j = y'$ for each $j \in C$ will succeed.

Because Shine is correct, then $\mathsf{Shine.Verify}(t, \mu, C, \{R_j\}_{j \in C})$ will output 1 and the same group commitment $R = \prod_{j \in C} R_j^{\lambda_j}$ will be computed regardless of the choice of C. Because $\mathsf{Shine.Gen}$ is deterministic, then after deriving $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$, $\mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y)$ will output the same (r_k, R_k) as derived in the first round of signing.

After deriving (r_k, R_k), all signers in a coalition C output valid signature shares z_k with respect to R and challenge $c = \mathsf{H}_3(\mathsf{pk}, R, m)$, where $z_k = r_k + c \cdot \mathsf{sk}_k^{(2)}$. The aggregated signature is then $\sigma = (R, z)$, where $z = \sum_{j \in C} z_j \cdot \lambda_j$. Because $\mathsf{sk} = \sum_{j \in C} \mathsf{sk}_j^{(2)} \lambda_j$, $\mathsf{pk} = g^{\mathsf{sk}} = g^{\sum_{j \in C} \mathsf{sk}_j^{(2)} \lambda_j}$, and $R = \prod_{j \in C} R_j^{\lambda_j} = g^{\sum_{j \in C} r_j \cdot \lambda_j}$, we have that $g^z = R \cdot \mathsf{pk}^c$, as required.

Unforgeability. We next demonstrate the unforgeability of Arctic.

Theorem 4. *Arctic is unforgeable against a PPT adversary $\mathcal{A}$ playing the static unforgeability game as shown in Figure 2 against Arctic, assuming $\mathcal{A}$ can make up to $t-1$ corruptions, the number of honest parties is at least t, the discrete logarithm assumption holds in the ROM, participants exchange messages over an authenticated channel, Shine is a secure VPSS, and where $(n, t) \in \mathbb{N}$ are such that $\binom{n-1}{t-1} = poly(n)$.*

Concretely, let $\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda)$ be the advantage of an adversary $\mathcal{D}$ against the discrete logarithm assumption. The advantage of $\mathcal{A}$ is bounded by

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{A},\mathsf{Arctic}}(\lambda, n, t, \mu) \leq \sqrt{q_r \mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda) + \frac{2(q_1 + q_3)}{q} + \frac{3q_r^2}{q}}$$

where $\mu \geq 2t - 1$ and $n \geq \mu$, and where $q_r = q_2 + q_3 + 2q_s + 1$, such that q_s is the number of times $\mathcal{A}$ is allowed to query the signing oracles, q_1 is the number of times $\mathcal{A}$ is allowed to query H_1, q_2 is the number of times that $\mathcal{A}$ is allowed to query H_2, and q_3 is the number of times $\mathcal{A}$ is allowed to query H_3.

To prove Theorem 4, we rely on two lemmas.

Lemma 2. *There exists an algorithm SIM which can simulate Arctic with respect to a discrete logarithm challenge to an adversary $\mathcal{A}$ playing against the unforgeability game as in Figure 2. In other words, $\mathcal{A}$ has negligible additional advantage in winning the unforgeability game when interacting with SIM that it does when playing against Arctic directly.*

Concretely, in the honest majority setting with authenticated channels, then

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda, n, t, \mu) \leq \mathsf{Adv}^{\mathrm{uf}}_{\mathsf{SIM},\mathcal{A}}(\lambda, n, t, \mu) + \frac{(q_1 + q_3)}{q} + \frac{q_r^2}{2q} \tag{10}$$

where $\mu \geq 2t - 1$ and $n \geq \mu$, and where (n, t) are such that $\binom{n-1}{t-1} = poly(n)$.

Lemma 3. *Given any PPT adversary $\mathcal{A}$ that wins the unforgeability game against Arctic and a simulating algorithm SIM which can simulate Arctic, then there exists a PPT DL adversary $\mathcal{D}$ that can use $\mathcal{A}$ as a subroutine to solve for its DL challenge.*

Concretely, in the honest majority setting with authenticated channels,

$$\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda) \geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{A},\mathsf{Arctic}}(\lambda, n, t, \mu)^2}{q_r} - \frac{2(q_1 + q_3)}{q} - \frac{3q_r^2}{q} \tag{11}$$

where $\mu \geq 2t - 1$ *and* $n \geq \mu$*, and* $q_r = q_2 + q_3 + 2q_s + 1$*, and where* $(n, t) \in \mathbb{N}$ *are such that* $\binom{n-1}{t-1} = poly(n)$*.*

We give the corresponding proofs for Theorem 4, Lemma 2, and Lemma 3 in Appendix C. We however next give a high-level proof intuition.

Proof Intuition. We give the proof in two parts. First, Lemma 2 shows that a simulating PPT algorithm SIM exists which can simulate Arctic to an unforgeability PPT adversary $\mathcal{A}$ with respect to a public key pk for which SIM does not know the corresponding secret key. Recall that we give the unforgeability game in Figure 2. We show that the advantage of $\mathcal{A}$ with respect to SIM is negligibly different than its advantage against Arctic.

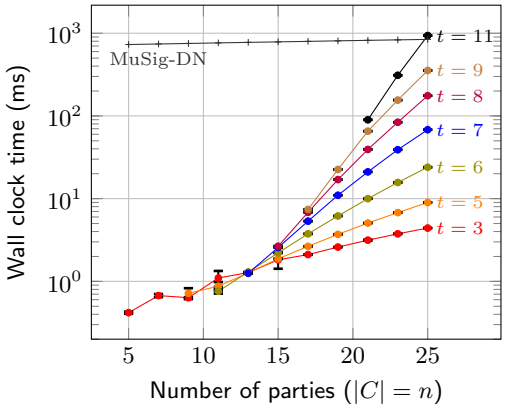
We prove Lemma 2 using a series of game hops, showing that because Shine is verifiable, unique, and pseudorandom, then SIM can simulate a signing session for a message m chosen by $\mathcal{A}$ with respect to pk. in a way that is indistinguishable to $\mathcal{A}$ from a real execution of the signing protocol. SIM does this by sampling (z, c) uniformly at random in $\mathcal{O}^{\mathsf{Sign}_1}$, deriving R as $R = g^z \cdot \mathsf{pk}^{-c}$, and then programming $\mathsf{H}_3(R, \mathsf{pk}, \mathsf{m})$ on c. Then, to simulate each honest party's commitment $R_i, i \in \mathsf{honest}$, SIM uses its knowledge of corrupt parties' Shine keys $\mathsf{sk}_j^{(1)}, j \in \mathsf{corrupt}$ to derive corrupt parties' nonces and commitments, by performing $(r_j, R_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j^{(1)}, y)$. SIM can then correctly derive each honest party's commitment $R_k, k \in \mathsf{honest}$ by performing polynomial interpolation in the exponent, with respect to R and the set $(R_j)_{j \in \mathsf{corrupt}}$.

Because Shine is verifiable, if $\mathcal{A}$ submits commitments $(R'_j)_{j \in C \cap \mathsf{corrupt}} \neq (R_j)_{j \in C \cap \mathsf{corrupt}}$ to $\mathcal{O}^{\mathsf{Sign}_2}$, SIM will output $\perp$ with the same probability as in the real scheme. Because Shine is unique, then R is fixed with respect to $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$, regardless of $\mathcal{A}$'s choice of coalition in the second round. Finally, because Shine is pseudorandom, then unless $\mathcal{A}$ can guess $\hat{\phi}$, where $\hat{\phi}$ is an honest party's portion of the replicated secret defined in $\mathsf{Shine.KeyGen}$, then SIM can simulate each honest party's $R_i, i \in \mathsf{honest}$ with negligible advantage to $\mathcal{A}$. We show that in order for $\mathcal{A}$ to break the pseudorandomness of Shine, it must guess a preimage of H_1, which it can do with negligible advantage less than q_1/q, where q_1 is the number of queries $\mathcal{A}$ is allowed to make to H_1.

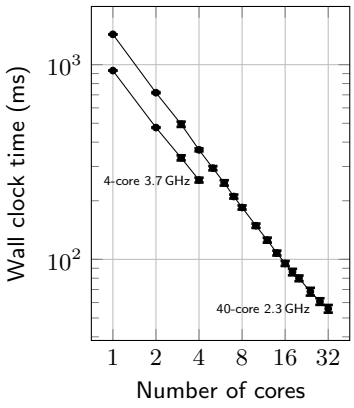
If $\mathcal{A}$ wins the unforgeability experiment against Arctic, $\mathcal{A}$ will output a forgery $\sigma^* = (R^*, z^*)$ corresponding to a message m^* that is valid under a challenge public key pk. Lemma 3 shows that given a successful PPT adversary $\mathcal{A}$, a PPT reduction $\mathcal{D}$ exists against the Discrete Logarithm assumption. In particular, we show that $\mathcal{D}$ can use $\mathcal{A}$ and SIM to solve for its discrete logarithm challenge. $\mathcal{D}$ accepts a group element $Y \in \mathbb{G}$ from its discrete logarithm challenger, and its goal is to output y such that $Y = g^y$. It uses the modified forking algorithm $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ shown in Figure 1 on the instance $X = (\mathsf{par}, Y, n, t, \mu)$, where n, t, μ are chosen by $\mathcal{D}$. Fork_m runs SIM twice, the first time on X and the set $\{h_1, \ldots, h_{q_r}\}$, which is the set of random oracle outputs that SIM will use to program its random oracle. On the second execution of SIM, SIM is again given X and the set $\{h_1, \ldots, h_{j-1}, h'_j, \ldots, h'_{q_r}\}$, where h_j corresponds to the challenge $c^* = H(R^*, \mathsf{pk}, m^*)$ for the forgery σ^* output by $\mathcal{A}$ in its first execution. Corollary 1 lower-bounds the probability that $\mathcal{A}$ will output a second forgery $\sigma^{**} = (R^*, z^{**})$ on the same random oracle query $c^{**} = H(R^*, \mathsf{pk}, m^*)$ as in its first iteration, where the programmed outputs $c^* = h_j, c^{**} = h'_j$ are such that $c^* \neq c^{**}$. $\mathcal{D}$ can use the two forgeries output by $\mathcal{A}$ in its first and second execution to then solve for the discrete logarithm solution $y = \frac{z^* - z^{**}}{c^* - c^{**}}$.

5.3 Extending Arctic and Shine to be Robust

Arctic as currently defined is not robust; if any party submits invalid commitments, then the output from $\mathsf{Shine.Agg}$ cannot be used. However, it is possible to extend Shine to be robust, therefore also ensuring that Arctic can likewise be extended. To do so in a secure manner, the robust extension would require additional players, along with a protocol to come to consensus about which players misbehaved.



(a) Single-core wall clock time for various parameter combinations for Arctic. The times shown are the sum of the computation times for Sign_1, Sign_2, and Combine. The computation time for MuSig-DN is shown for comparison.



(b) Scaling experiment, showing the wall clock time for the largest configuration $(n, |C|, t) = (25, 25, 11)$ shown in Figure 7a as we increase the number of CPU cores.

Fig. 7: Experimental results for Arctic

In particular, by requiring that the minimum number of participants μ be of size at least $\mu \geq 3t - 2$, Shine and Arctic can be securely used in a robust manner. The requirement that $\mu \geq 3t - 2$ is referred to as the *honest supermajority* setting. In this setting, Shine.Verify can both detect any inconsistencies as well as identify the misbehaving players. This property could likewise allow for extending Arctic to support robustness, under the same assumption that $\mu \geq 3t - 2$.

We present more details on how robustness could be achieved in Appendix D.

6 Performance Analysis of Arctic

In this section, we analyze the performance of Arctic. In terms of the number of rounds of communication, Arctic matches the state of the art, with two, and it sends significantly less bandwidth per signature, at 65 bytes per participant. Therefore, we focus on the computational complexity.

There are two sources of potentially expensive computation: the two calls to Shine.Gen (one in each of Sign_1 and Sign_2), and the call to Shine.Verify in Sign_2. Which one dominates depends on the parameters n, t, and the number of participants in the signing protocol. Recall that C is a set representing the identifiers of participants in a particular signing session. Although the minimum number of participants required for signing is $|C| \geq \mu \geq 2t - 1$, we assume $|C| = n$ for this analysis, to give an upper performance bound. As such, performance will be even better when $\mu \leq |C| < n$.

Shine.Gen primarily performs $\delta = \binom{n-1}{t-1}$ hash computations, field multiplications, and field additions, as seen in Equation 4 (recalling that the $L'_{a_i}(k)$ values can be precomputed). Shine.Verify computes $\ell = |C| - t$, $|C|$-way multiexponentiations. When t is small, we expect the Shine.Verify cost to dominate, and for larger t, the Shine.Gen cost should dominate.

To concretely evaluate the performance of Arctic, we implemented it in Rust.[3] We ran our implementation over all allowable combinations of parameters $t \geq 2$, $2t - 1 \leq |C| \leq n \leq 25$, using a 4-core 3.7 GHz Intel E-2374G CPU. We measured the computation time for each of Sign_1, Sign_2, and Combine, averaged over 10 signatures, for each parameter combination. In Figure 7a, we show the (single-core) total runtime of Sign_1, Sign_2, and Combine for various combinations of parameters. We concretely measure Shine.Gen to take

[3] Our code is available at https://git-crysp.uwaterloo.ca/iang/arctic/.

around 0.24δ microseconds for each of its two invocations, and Shine.Verify to take around $7\ell|C|$ microseconds. For $t \leq 4$, the latter dominates, for $t = 5$, they are roughly comparable, and for $t \geq 6$, the former quickly dominates.

For comparison, we also show the computational time for MuSig-DN, but *only* the zero-knowledge proof and verification components of their algorithm. We ran their code [43] on our same machine to obtain these figures. We can see that for $n \leq 20$, Arctic is more than an order of magnitude faster than MuSig-DN, and for $n \leq 10$, it is three orders of magnitude faster.

As seen in Figure 7a, the computation time for the $(n, |C|, t) = (25, 25, 11)$ parameter combination, where $\delta = \binom{24}{10} = 1961256$ is around 940 ms, almost all of which is spent in Shine.Gen computing Equation 4. However, we observe that Equation 4 computes the sum of δ independent terms, and so is highly amenable to parallelization, which we also implemented and measured. We ran this scaling experiment both on the above machine, and also on a 40-core 2.3 GHz Intel 8380 CPU. The results are shown in Figure 7b. Although the slower clock speed of the 40-core CPU puts it at a disadvantage for smaller numbers of cores, we can see almost linear scaling for both CPUs. Using all 4 cores, the 4-core CPU sees a speedup of 3.69× for Shine.Gen and 3.65× in total time, while using 32 cores, the 40-core CPU sees a speedup of 27.9× for Shine.Gen and 25.7× in total time. Beyond 32 cores, we observed diminishing returns.

7 Conclusion

In this work, we presented Arctic, a deterministic and stateless threshold Schnorr signature scheme for the honest majority setting. By not requiring zero-knowledge proofs of verifiable random functions, Arctic is simpler than previous deterministic threshold Schnorr schemes, and for small to moderate sized groups of signers, Arctic is one to three orders of magnitude faster.

References

1. T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 843–862. IEEE Computer Society, 2017. doi:10.1109/SP.2017.15.
2. D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991. doi:10.1007/3-540-46766-1_34.
3. M. Bellare, E. C. Crites, C. Komlo, M. Maller, S. Tessaro, and C. Zhu. Better than advertised security for non-interactive threshold signatures. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022*, volume 13510 of *LNCS*, pages 517–550. Springer, 2022.
4. M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 390–399. ACM, 2006.
5. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. doi:10.1007/11761679_25.
6. F. Benhamouda, E. Boyle, N. Gilboa, S. Halevi, Y. Ishai, and A. Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In K. Nissim and B. Waters, editors, *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021*, volume 13043 of *LNCS*, pages 129–161. Springer, 2021. doi:10.1007/978-3-030-90453-1_5.
7. F. Benhamouda, T. Lepoint, J. Loss, M. Orrù, and M. Raykova. On the (in)security of ROS. In A. Canteaut and F. Standaert, editors, *EUROCRYPT 2021, Zagreb, Croatia, October 17-21, 2021*, volume 12696 of *LNCS*, pages 33–53. Springer, 2021.

8. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In Y. Desmedt, editor, *PKC 2003, Miami, FL, USA, January 6-8, 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, 2003.
9. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.
10. C. Bonte, N. P. Smart, and T. Tanguy. Thresholdizing HashEdDSA: MPC to the Rescue. *Int. J. Inf. Sec.*, 20(6):879–894, 2021. doi:10.1007/s10207-021-00539-6.
11. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators from ring-lpn. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020*, volume 12171 of *Lecture Notes in Computer Science*, pages 387–416. Springer, 2020. doi:10.1007/978-3-030-56880-1_14.
12. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018. doi:10.1109/SP.2018.00020.
13. R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1769–1787. ACM, 2020.
14. H. Chu, P. Gerhart, T. Ruffing, and D. Schröder. Practical schnorr threshold signatures without the algebraic group model. In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023*, volume 14081 of *LNCS*, pages 743–773. Springer, 2023. doi:10.1007/978-3-031-38557-5_24.
15. D. Connolly, C. Komlo, I. Goldberg, and C. Wood. Two-round threshold Schnorr signatures with FROST, 2022. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/.
16. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005. doi:10.1007/978-3-540-30576-7_19.
17. E. C. Crites, C. Komlo, and M. Maller. Fully adaptive schnorr threshold signatures. In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023*, volume 14081 of *LNCS*, pages 678–709. Springer, 2023. doi:10.1007/978-3-031-38557-5_22.
18. I. Damgård, T. P. Jakobsen, J. B. Nielsen, J. I. Pagter, and M. B. Østergaard. Fast threshold ECDSA with honest majority. *J. Comput. Secur.*, 30(1):167–196, 2022. doi:10.3233/JCS-200112.
19. Y. Dodis. Efficient construction of (distributed) verifiable random functions. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2003. doi:10.1007/3-540-36288-6_1.
20. M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven, and I. Stepanovs. On the security of two-round multi-signatures. In *SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1084–1101. IEEE, 2019.
21. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO 1986, Santa Barbara, California, USA, 1986*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.
22. M. Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168. Springer, 2005.
23. D. Galindo, J. Liu, M. Ordean, and J. Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 88–102. IEEE, 2021. URL: https://doi.org/10.1109/EuroSP51992.2021.00017, doi:10.1109/EUROSP51992.2021.00017.
24. F. Garillot, Y. Kondi, P. Mohassel, and V. Nikolaenko. Threshold schnorr with stateless deterministic signing from standard assumptions. In T. Malkin and C. Peikert, editors, *CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021*, volume 12825 of *LNCS*, pages 127–156. Springer, 2021. doi:10.1007/978-3-030-84242-0_6.
25. R. Gennaro and S. Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1179–1194. ACM, 2018.
26. R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In U. M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application*

of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding, volume 1070 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 1996. doi:10.1007/3-540-68339-9_31.
27. R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. *Inf. Comput.*, 164(1):54–84, 2001.
28. R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.
29. R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. Robust and efficient sharing of RSA functions. *J. Cryptol.*, 20(3):393, 2007.
30. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.
31. C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017*, volume 10624 of *LNCS*, pages 598–628. Springer, 2017. doi:10.1007/978-3-319-70694-8_21.
32. M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/ecjc.4430720906, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ecjc.4430720906, doi:https://doi.org/10.1002/ecjc.4430720906.
33. S. Jarecki, H. Krawczyk, and J. Resch. Threshold partially-oblivious PRFs with applications to key management. 2018. URL: https://eprint.iacr.org/2018/733.
34. M. Jawurek, F. Kerschbaum, and C. Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 955–966. ACM, 2013. doi:10.1145/2508859.2516662.
35. C. Komlo and I. Goldberg. FROST: flexible round-optimized Schnorr threshold signatures. In O. Dunkelman, M. J. J. Jr., and C. O'Flynn, editors, *Selected Areas in Cryptography - SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, 2020. doi:10.1007/978-3-030-81652-0_2.
36. Y. Kondi. Personal communication, 2024.
37. Y. Kondi, C. Orlandi, and L. Roy. Two-round stateless deterministic two-party schnorr signatures from pseudorandom correlation functions. In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023*, volume 14081 of *Lecture Notes in Computer Science*, pages 646–677. Springer, 2023. doi:10.1007/978-3-031-38557-5_21.
38. Y. Lindell. Simple three-round multiparty Schnorr signing with full simulatability. Cryptology ePrint Archive, Report 2022/374, 2022. https://ia.cr/2022/374.
39. N. Makriyannis. On the classic protocol for mpc schnorr signatures. Cryptology ePrint Archive, Paper 2022/1332, 2022. https://eprint.iacr.org/2022/1332. URL: https://eprint.iacr.org/2022/1332.
40. N. Makriyannis, O. Yomtov, and A. Galansky. Practical Key-Extraction Attacks in Leading MPC Wallets. Cryptology ePrint Archive, Paper 2023/1234, 2023. URL: https://eprint.iacr.org/2023/1234.
41. S. Micali, M. O. Rabin, and S. P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 120–130. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814584.
42. M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and kdcs. In J. Stern, editor, *EUROCRYPT '99*, volume 1592 of *LNCS*, pages 327–346. Springer, 1999.
43. J. Nick. Switch bulletproof example to purify. https://github.com/jonasnick/secp256k1-zkp/blob/bulletproof-musig-dn-benches/examples/bulletproof.c, 2020.
44. J. Nick, T. Ruffing, Y. Seurin, and P. Wuille. MuSig-DN: Schnorr Multi-Signatures with Verifiably Deterministic Nonces. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1717–1731. ACM, 2020. doi:10.1145/3372297.3417236.
45. C. Orlandi, P. Scholl, and S. Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In A. Canteaut and F. Standaert, editors, *EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings*, volume 12696 of *LNCS*, pages 678–708. Springer, 2021. doi:10.1007/978-3-030-77870-5_24.
46. D. Pointcheval and J. Stern. Security proofs for signature schemes. In U. M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 1996. doi:10.1007/3-540-68339-9_33.

47. D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *J. Cryptol.*, 13(3):361–396, 2000.
48. T. Ristenpart and S. Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010. URL: https://www.ndss-symposium.org/ndss2010/when-good-randomness-goes-bad-virtual-machine-reset-vulnerabilities-and-hedging-deployed.
49. T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder. ROAST: robust asynchronous schnorr threshold signatures. In H. Yin, A. Stavrou, C. Cremers, and E. Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2551–2564. ACM, 2022. doi:10.1145/3548606.3560583.
50. C. Schnorr. Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174, 1991.
51. C. Schnorr. Enhancing the security of perfect blind dl-signatures. *Inf. Sci.*, 176(10):1305–1320, 2006. URL: https://doi.org/10.1016/j.ins.2005.04.007, doi:10.1016/J.INS.2005.04.007.
52. A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
53. D. R. Stinson and R. Strobl. Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates. In V. Varadharajan and Y. Mu, editors, *ACISP 2001, Sydney, Australia, July 11-13, 2001*, volume 2119 of *LNCS*, pages 417–434. Springer, 2001.

A Additional Performance Estimates

Here, we expand on our performance estimates given in Table 1. We do not give exact computation estimates, but instead give estimates for computation that dominate. For bandwidth, we estimate for a 256-bit elliptic curve, and so estimate scalers as 32 bytes and group elements as 33 bytes.

MuSig-DN [44]. Nick et al. [44] present the proof size as 1124 bytes, which agrees with our measurements. Further, each participant sends one group element in round one, and outputs one field element in round three, yielding the total of 1189 bytes we report in Table 1.

While Nick et al. give total computational overhead for MuSig-DN for a 256-bit curve, we give more detail as to computation that dominates when performing signing operations.

The computational overhead of MuSig-DN is dominated by the cost of generating and verifying Bulletproof [12] proofs. For each signing operation, each participant must generate one proof and verify $(n-1)$ other proofs. While the computational overhead of Bulletproofs scales linearly relative to the size of the circuit, batch verification and multi-scalar multiplication allow for improved performance.

Proofs in MuSig-DN require a circuit of 2030 multiplication gates [44], and estimates are given with respect to a 256-bit elliptic curve group. Naively, this overhead would translate into roughly $2030n$ operations for each signer, due to the requirement that each signer generates a proof and verifies $n-1$ other proofs.

However, when considering speedups offered by multi-scalar multiplications and batch verification, the overhead for a signer becomes roughly 2030 operations, plus additional (small) terms linear in n. When generating a MuSig-DN proof using Bulletproofs, we estimate that the dominant computational overhead is roughly $6 \cdot 2030$ multi-scalar multiplications, to commit to the circuit constraints and additional factors. For batch verifying $n-1$ MuSig-DN proofs, we estimate the dominant computational overhead to the verifier is $2030 + 24(n-1)$ scalar multiplications.

GKMN21 [24]. Garillot et al. [24] give computational and bandwidth estimates for a single sender and receiver, but each player will be involved in $t-1$ concurrent sending and receiving sessions with all other players. As such, the estimate we give in Table 1 is for the total sending and receiving overhead for each signer in a signing session with t participants.

Arctic. For bandwidth, each party sends one group element and one field element as the output to the first signing query, and then outputs a single field element in the second signing round.

For computation, each party performs $\mathsf{Shine.Gen}$ once for each signing round, resulting in $\binom{n-1}{t-1}$ field and hash operations, and one group multiplication, for each execution. In the second round of signing, each party performs $\mathsf{Shine.Gen}$ again, for another $\binom{n-1}{t-1}$ field and hash operations, and one group multiplication, and also $\mathsf{Shine.Verify}$ and $\mathsf{Shine.Agg}$, which require $2t^2 - t$ group multiplications.

B Security of Shine

We define the security of Shine in Section 4.3; we give the corresponding proofs here.

Proof (Of Theorem 1). Recall that in the verifiability game, the adversary is required to provide an input w, a coalition $C \subseteq [n]$, and a set $(D_i)_{i \in C \cap \mathsf{corrupt}}$, such that $(D_i)_{i \in C \cap \mathsf{corrupt}}$ does not equal the set $(D'_i)_{i \in C \cap \mathsf{corrupt}}$, where $(d'_i, D'_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i, w)$ is obtained honestly. The environment derives the values $(d_j, D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w), j \in C \cap \mathsf{honest}$ for honest participants, reflecting the requirement that participants' messages are received over authenticated channels. The adversary wins when $\mathsf{Shine.Verify}(t, \mu, C, S)$ outputs 1, where $S = (D_j)_{j \in C}$.

For $\mathsf{Shine.Verify}(t, \mu, C, S)$ to output 1, then Equation 6 must hold. But for Equation 6 to hold, then $(D_j)_{j \in C}$ must define a degree $t-1$ polynomial in the exponent. However, this polynomial is information-theoretically defined by the set of honest participant commitments $(D_j)_{j \in C \cap \mathsf{honest}}$, because we assume $|\mathsf{honest}| \geq t$, and a degree $t-1$ polynomial is uniquely defined by t points.

Hence, when corrupted participants do not honestly follow the protocol, then the resulting polynomial committed to in S will be of higher degree than $t-1$. As such, Shine is information-theoretically verifiable. □

$\mathsf{SimGen}(w, d, \mathsf{corrupt}, n, \{\mathsf{sk}_j\}_{j\in\mathsf{corrupt}})$

$D \leftarrow g^d$

$(d_j, D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w)$, for $j \in \mathsf{corrupt}$

// *This step is possible because Shine is verifiable,*

// *and so it is guaranteed that the correct* $(d_j, D_j), j \in \mathsf{corrupt}$ *are derived*

Define $f(x) = d \cdot L_0(x) + \sum_{j\in\mathsf{corrupt}} d_j \cdot L_j(x)$

// $L_i(x)$ *is the Lagrange polynomial defined by* $\mathsf{corrupt} \cup \{0\}$

$\mathsf{honest} = [n] \setminus \mathsf{corrupt}$

for $i \in \mathsf{honest}$ **do**

  Derive $d_i \leftarrow f(i)$

  $D_i \leftarrow g^{d_i}$

return $(D, \{(d_i, D_i)\}_{i\in\mathsf{honest}})$

Fig. 8: Simulating algorithm SimGen for Shine.

Proof (Of Theorem 2). In the uniqueness game given in Figure 3, the adversary is required to provide an input w and the tuples $(C, (D_i)_{i\in C\cap\mathsf{corrupt}}), (C', (D'_i)_{i\in C'\cap\mathsf{corrupt}})$. The environment then derives honest parties' contributions $(D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w)$, again reflecting the authenticated channel, and the environment derives the commitment sets $S_1 = (D_j)_{j\in C\cap\mathsf{honest}} \cup (D_i)_{i\in C\cap\mathsf{corrupt}}$ and $S_2 = (D_j)_{j\in C'\cap\mathsf{honest}} \cup (D'_i)_{i\in C'\cap\mathsf{corrupt}}$. The adversary wins if both sets are valid for their respective coalitions, and yet the output of Shine.Agg outputs different combined commitments.

However, for $\mathsf{Shine.Verify}(C, S_1)$ and $\mathsf{Shine.Verify}(C', S_2)$ to both output 1, then Equation 6 must hold. But for Equation 6 to hold, then S_1 and S_2 must each define a degree $t-1$ polynomial. Further, because we assume at least t honest participants, then S_1 and S_2 must agree on at minimum t points.

As such, when both $\mathsf{Shine.Verify}(C, S_1)$ and $\mathsf{Shine.Verify}(C, S_2)$ both output 1, then the sets S_1 and S_2 must define the same degree $t-1$ polynomial. Because Shine.Agg simply performs polynomial interpolation in the exponent to output the commitment to the constant term of the polynomial, then information-theoretically, it must be the case that $\mathsf{Shine.Agg}(S_1) = \mathsf{Shine.Agg}(S_2)$. Therefore, uniqueness likewise holds information-theoretically for Shine. □

Proof (Of Theorem 3). We prove Theorem 3 by a sequence of games.

Game 0. This is the peudorandomness game as defined in Figure 5 instantiated with Shine, when $b = 0$. Let W_0 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 0.

Game 1. Let $\mathsf{sk}_{\mathsf{corrupt}} = \cup_{i\in\mathsf{corrupt}}\mathsf{sk}_i$, and let $\mathsf{sk}_{\mathsf{honest}} = \cup_{j\in\mathsf{honest}}\mathsf{sk}_j$. Then $\mathsf{sk}_{\mathsf{honest}} \setminus \mathsf{sk}_{\mathsf{corrupt}}$ is necessarily non-empty, so let $\hat{\phi}$ be such that $(\cdot, \hat{\phi}) \in \mathsf{sk}_{\mathsf{honest}} \setminus \mathsf{sk}_{\mathsf{corrupt}}$.

Note that this $\hat{\phi}$ will be random in $\mathbb{Z}_q$ and unknown to $\mathcal{A}$. Then the only difference between Game 0 and Game 1 is that if $\mathcal{A}$ queries H on any input $(\hat{\phi}, \cdot)$, the environment aborts.

Difference between Game 1 and Game 0. When $\mathcal{A}$ is allowed to make up to q_h queries to H, then the probability the environment aborts is at most q_h/q. Let W_1 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 1. Then

$$|\Pr[W_1] - \Pr[W_0]| \leq \frac{q_h}{q} \tag{12}$$

Game 2. This is the pseudorandomness game as in Figure 5, when $b = 1$, and where SimGen is as shown in Figure 8, along with the abort condition of Game 1.

On start, the environment initializes a table $Q \leftarrow \emptyset$. Then, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Gen}}$ on input (k, w), instead of performing $\mathsf{Shine.Gen}$ honestly, the environment first checks if $Q[w] \neq \perp$; if the check holds, it parses $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow Q[w]$ and returns (d_k, D_k).

Otherwise, the environment samples $d \xleftarrow{\$} \mathcal{P}$, and then obtains $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(w, d, \mathsf{corrupt}, n, \{\mathsf{sk}_j\}_{j \in \mathsf{corrupt}})$, where SimGen is defined as in Figure 8. Note that SimGen requires that Shine to be verifiable, and hence that $|\mathsf{honest}| \geq t$. The environment updates $Q[w] = (D, \{(d_i, D_i)\}_{i \in \mathsf{honest}})$, and then returns (d_k, D_k).

Analysis of SimGen Because Game 1 aborts on the condition that the adversary guesses the honest participants' secret key, in Game 2, the adversary must win with knowledge of only $t-1$ secret keys. However, because we assume that the number of honest parties is at least t, then the honest parties will completely determine the combined d.

Furthermore, because Shine is verifiable, then SimGen can derive the corrupt parties' contributions $(d_j, D_j), j \in \mathsf{corrupt}$ before it determines honest parties' contributions. As such, corrupt parties cannot adaptively choose their contributions after the fact to influence the resulting $D \leftarrow \mathsf{Shine.Combine}(t, \mu, \{D_i\}_{i \in C^*})$. Hence, the simulation by SimGen is perfect.

Difference between Game 2 and Game 1. In Game 1, $\mathcal{O}^{\mathsf{Gen}}(k, C, w)$ returns $(f(k), g^{f(k)})$ where $f(x) = \sum_{i \in [\delta]} \mathsf{H}(\phi_i, w) \cdot L'_{a_i}(x)$ defined as in Equation 4. Since $\mathsf{H}(\hat{\phi}, w)$ is unknown to $\mathcal{A}$, $f(x)$ is a random degree-$(t-1)$ polynomial that passes through $(j, y_j)_{j \in \mathsf{corrupt}}$.

In Game 2, $\mathcal{O}^{\mathsf{Gen}}(k, C, w)$ returns $(f(k), g^{f(k)})$, where $f(x) = d \cdot L_0(x) + \sum_{j \in \mathsf{corrupt}} d_j \cdot L_j(x)$. Since d is random, this is also a random degree-$(t-1)$ polynomial that passes through $(j, y_j)_{j \in \mathsf{corrupt}}$.

There is no difference between the distributions of these outputs. Let W_2 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 2. Then

$$|\Pr[W_2] - \Pr[W_1]| = 0 \tag{13}$$

Game 3. Game 3 is identical to Figure 5 instantiated with Shine, when $b = 1$.

Difference between Game 3 and Game 2. The only difference between Game 3 and Game 2 is that Game 2 has the abort condition, but Game 3 does not. Let W_3 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 3. Then as above,

$$|\Pr[W_3] - \Pr[W_2]| \leq \frac{q_h}{q} \tag{14}$$

Finishing the Proof. Combining Equations 12–14 yields

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{A},\mathsf{Shine}}(\lambda, n, t, \mu) &= \left|\Pr[\mathsf{Exp}^{\mathrm{psdr}}_{\mathcal{A},\mathsf{Shine}}(\lambda, n, t, \mu) = 1] - 1/2\right| \\
&= \left|\Pr[b' = 0 \wedge b = 0] + \Pr[b' = 1 \wedge b = 1] - 1/2\right| \\
&= \left|\frac{1}{2}(1 - \Pr[W_0]) + \frac{1}{2}\Pr[W_3] - \frac{1}{2}\right| \\
&= \left|\frac{1}{2}(\Pr[W_3] - \Pr[W_0])\right| \leq \frac{q_h}{q},
\end{aligned}$$

which gives Equation 9. This completes the proof. □

C Security of Arctic

In Section 5.2, we present Theorem 4 to demonstrate the unforgeability of Arctic, and Lemma 2 and Lemma 3 to support the result. We give the corresponding proofs now.

Proof (Of Lemma 2). We prove Lemma 2 by a sequence of games.

Game 0. This is the unforgeability game as defined in Figure 2, instantiated with Arctic. Let $\mathcal{A}$ be an adversary playing against the unforgeability game.

Let W_0 be the event that $\mathcal{A}$ wins Game 0. Then, the advantage of $\mathcal{A}$ is simply

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda) = \Pr[W_0] \tag{15}$$

Game 1. The only difference between Game 0 and Game 1 is as follows. If $\mathcal{A}$ queries H_2 or H_3 on any input that produces a colliding output with another query, then the environment aborts.

Difference between Game 1 and Game 0. Let W_1 be the event that $\mathcal{A}$ wins Game 1. Because we model H_2 and H_3 as random oracles, then

$$|\Pr[W_1] - \Pr[W_0]| \leq \frac{q_r^2}{2q} \tag{16}$$

Game 2. The only difference between Game 2 and Game 1 is as follows. If $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on an input (k, m, C, S_1) where $S_1 = (y_i, R_i)_{i \in C}$, such that the following relation holds:

$$\begin{aligned} y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m});\ y = y_i, i \in C,\ \text{and} \\ |C \cap \mathsf{honest}| \geq t, \\ \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y) = (r_i', R_i') \quad \forall i \in C \text{ and} \\ \mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j \in C}) = 1,\ \text{but} \\ S_1 \neq (y_i, R_i')_{i \in C} \end{aligned} \tag{17}$$

then the environment aborts.

Reduction to Verifiability of Shine. We now define a reduction $\mathcal{B}_2$ that uses $\mathcal{A}$ as a subroutine when it plays against the verifiability game of Shine.

To begin, $\mathcal{B}_2$ receives as input from the VPSS verifiability game as shown in Figure 4 the set of Shine secret keys $(\mathsf{sk}_i^{(1)})_{i \in \mathsf{corrupt}}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j \in [n]}, (\mathsf{sk}_i)_{i \in \mathsf{corrupt}})$.

To simulate signing, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_2$ derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. $\mathcal{B}_2$ then queries its own VPSS verifiability oracle $\mathcal{O}^{\mathsf{Gen}}$ on (k, y) receiving (d_k, D_k) as the output. It sets $R_k = D_k$, and outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, \mathsf{m}, C, (y_j, R_j')_{j \in C})$, $\mathcal{B}_2$ first performs $\mathsf{Shine.Verify}$ on the inputs, outputting $\perp$ if the input is invalid. Otherwise, $\mathcal{B}_2$ then derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly.

$\mathcal{B}_2$ then derives each corrupted party's nonce and commitment honestly, by performing $(r_i', R_i') \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y)$, for $i \in \mathsf{corrupt} \cap C$. If any $(R_i \neq R_i')$, for $i \in \mathsf{corrupt} \cap C$, then $\mathcal{B}_2$ terminates the execution of $\mathcal{A}$. $\mathcal{B}_2$ then submits $(y, C, (R_i)_{i \in \mathsf{corrupt} \cap C})$ as its own output to the VPSS verifiability experiment.

Otherwise, $\mathcal{B}_2$ simply simulates by querying $\mathcal{O}^{\mathsf{Gen}}$ for each honest party $\mathsf{honest} \cap C$ on input y, to re-derive (r_j, R_j) for $j \in \mathsf{honest} \cap C$. $\mathcal{B}_2$ then follows the remainder of the protocol honestly, outputting its signature share z_k.

The simulation by $\mathcal{B}_2$ is perfect because the simulation by the VPSS oracle $\mathcal{O}^{\mathsf{Gen}}$ is perfect. Further, the early termination of $\mathcal{A}$ has no consequence on the indistinguishability of the simulation.

Difference between Game 2 and Game 1. The additional advantage to $\mathcal{A}$ in Game 2 is upper-bounded by the advantage that $\mathcal{B}_2$ wins its VPSS verifiability game against Shine. Let W_2 be the event that $\mathcal{A}$ wins in Game 2, and let $\mathsf{Adv}^{\mathrm{verf}}_{\mathcal{B}_2,\mathsf{Shine}}$ be the advantage that $\mathcal{B}_2$ has in the verifiability experiment against Shine. Because Shine is information-theoretically verifiable in the honest majority setting, and Arctic likewise assumes an honest majority then

$$|\Pr[W_2] - \Pr[W_1]| \leq \mathsf{Adv}^{\mathrm{verf}}_{\mathcal{B}_2,\mathsf{Shine}} = 0 \tag{18}$$

Game 3. The only difference between Game 2 and Game 3 is as follows. If $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on the inputs (m, C, S) and (m, C', S') such that $S = (y, R_i)_{i\in C}$ and $S' = (y, R'_i)_{i\in C'}$ and $(C, S) \neq (C', S')$, but the following relation holds:

$$\begin{aligned} y &\leftarrow \mathsf{H}_2(\mathsf{pk}, m) \\ (r_j, R_j) &\leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j^{(1)}, y), j \in \mathsf{honest} \cap (C \cup C') \\ |C \cap \mathsf{honest}| &\geq t, \quad |C' \cap \mathsf{honest}| \geq t \\ \mathsf{Shine.Verify}(t, \mu, C, (R_i)_{i\in C}) = 1, &\mathsf{Shine.Verify}(t, \mu, C', (R'_i)_{i\in C}) = 1, \text{ but} \\ \mathsf{Shine.Agg}(t, \mu, C, (R_i)_{i\in C}) &\neq \mathsf{Shine.Agg}(t, \mu, C', (R'_i)_{i\in C'}) \end{aligned} \tag{19}$$

then the environment aborts.

Reduction to Uniqueness of Shine. We now define a reduction $\mathcal{B}_3$ that uses $\mathcal{A}$ as a subroutine when it plays against the uniqueness game of Shine.

To begin, $\mathcal{B}_3$ receives as input from the VPSS uniqueness experiment the set of Shine secret keys $(\mathsf{sk}_i^{(1)})_{i\in\mathsf{corrupt}}$. $\mathcal{B}_3$ then generates a table Q to maintain queries by $\mathcal{A}$ to $\mathcal{O}^{\mathsf{Sign}_2}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j\in[n]}, (\mathsf{sk}_i)_{i\in\mathsf{corrupt}})$.

To simulate signing, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_3$ first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. It then queries its own VPSS uniqueness oracle $\mathcal{O}^{\mathsf{Gen}}$ on the input (k, y), receiving (d_k, D_k) as the output. It sets $R_k = D_k$, and then outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, \mathsf{m}, C, (y_j, R_j)_{j\in C})$, $\mathcal{B}_3$ again derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. After checking that $y = y_i$ for each i and that $k \in C$, it then updates Q to be $Q \leftarrow Q \cup \{(\mathsf{m}, C, y, \{R_j\}_{j\in C})\}$. $\mathcal{B}_3$ then checks Q to see if any two entries $(\mathsf{m}, C, y, \{R_j\}_{j\in C})$, $(\mathsf{m}, C', y, \{R'_j\}_{j\in C'})$ exist such that Equation 19 holds.

If the check succeeds, then $\mathcal{B}_3$ terminates the execution of $\mathcal{A}$. $\mathcal{B}_3$ then outputs $(y, (C, (R_j)_{j\in C\cap\mathsf{corrupt}}), (C', (R'_j)_{j\in C'\cap\mathsf{corrupt}}))$ as its own output to the VPSS uniqueness game.

If the check fails, then $\mathcal{B}_3$ again queries its own VPSS uniqueness oracle $\mathcal{O}^{\mathsf{Gen}}$ on the input (k, y), receiving (d_k, D_k) as the output. $\mathcal{B}_3$ then sets $r_k = d_k$ and $R_k = D_k$. It follows the rest of the protocol honestly.

The simulation by $\mathcal{B}_3$ is perfect because the simulation by the VPSS oracle $\mathcal{O}^{\mathsf{Gen}}$ is perfect. Further, the early termination of $\mathcal{A}$ has no consequence on the indistinguishability of the simulation.

Difference between Game 3 and Game 2. The additional advantage to $\mathcal{A}$ in Game 2 is upper-bounded by the advantage that $\mathcal{B}_3$ wins its VPSS uniqueness game against Shine. Let W_3 be the event that $\mathcal{A}$ wins in Game 3, and let $\mathsf{Adv}^{\mathrm{unq}}_{\mathcal{B}_3,\mathsf{Shine}}$ be the advantage that $\mathcal{B}_3$ has in the verifiability experiment against Shine. Because Shine is information-theoretically unique in the honest majority setting with authenticated channels, then

$$|\Pr[W_3] - \Pr[W_2]| \leq \mathsf{Adv}^{\mathrm{unq}}_{\mathcal{B}_3,\mathsf{Shine}} = 0 \tag{20}$$

Game 4. The only differences between Game 4 and Game 3 are as follows. On initialization, the environment initializes a table $Q \leftarrow \emptyset$.

Let $\mathsf{sk}^{(1)}_{\mathsf{corrupt}} = \cup_{i \in \mathsf{corrupt}} \mathsf{sk}^{(1)}_i$, and let $\mathsf{sk}^{(1)}_{\mathsf{honest}} = \cup_{j \in \mathsf{honest}} \mathsf{sk}^{(1)}_j$. Let $\hat{\phi}$ be such that $(\cdot, \hat{\phi}) \in \mathsf{sk}^{(1)}_{\mathsf{honest}} \setminus \mathsf{sk}^{(1)}_{\mathsf{corrupt}}$. If $\mathcal{A}$ queries H_1 on $\hat{\phi}$, then the environment aborts.

Otherwise, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on party $k \in \mathsf{honest}$ and message m, instead of producing each party's nonce using Shine, the environment first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. It then checks if $Q[y] \neq \perp$; if so, it parses $(r_i, R_i)_{i \in \mathsf{honest}} \leftarrow Q[y]$, and returns (y, R_k). Otherwise, it does the following:

1. Sample $r \xleftarrow{\$} \mathbb{Z}_q$.
2. Use the algorithm SimGen as defined by Shine in Figure 8 to generate $(R, \{(r_i, R_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(y, r, \mathsf{corrupt}, n, \{\mathsf{sk}^{(1)}_i\}_{i \in \mathsf{corrupt}})$.
3. Set $Q[y] = (r_i, R_i)_{i \in \mathsf{honest}}$.
4. Return (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on party $k \in \mathsf{honest}$, message m, coalition C, and set of commitments $(R_i)_{i \in C}$, the environment first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. It then checks if $Q[y] = \perp$; if so, it returns $\perp$. Otherwise, it parses $(r_i, R_i)_{i \in \mathsf{honest}} \leftarrow Q[y]$. It follows the protocol honestly to derive z_k, and outputs z_k.

Reduction to Pseudorandomness of Shine. We now define a reduction $\mathcal{B}_4$ that uses $\mathcal{A}$ as a subroutine when it plays against the pseudorandomness game of Shine. We assume without loss of generality that $|\mathsf{corrupt}| = t - 1$.

On initialization, $\mathcal{B}_4$ receives as input from the VPSS pseudorandomness experiment the set of Shine secret keys $(\mathsf{sk}^{(1)}_i)_{i \in \mathsf{corrupt}}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}^{(1)}_i, \mathsf{sk}^{(2)}_i, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j \in [n]}, (\mathsf{sk}_i)_{i \in \mathsf{corrupt}})$.

When $\mathcal{A}$ queries H_1 on inputs (ϕ_i, y), $\mathcal{B}_4$ does the following:

1. Sets k be the identifier of an honest party. Note that all of $\mathsf{sk}^{(1)}_k$ is known to $\mathcal{B}_4$ except the entry $(a_i, \hat{\phi})$ where $a_i = \mathsf{corrupt}$.
2. Queries $\mathcal{O}^{\mathsf{Gen}}$ on input (k, y), receiving in return (d_k, D_k).
3. Checks if the following relation holds:

$$\mathsf{H}_1(\phi_i, y) \stackrel{?}{=} d_k - \frac{\sum_{(a_\ell, \phi_\ell) \in \mathsf{sk}^{(1)}_k, \ell \neq i} \mathsf{H}_1(\phi_\ell, y) \cdot L'_{a_\ell}(k)}{L'_{a_i}(k)} \tag{21}$$

   where $L'_{a_\ell}(x)$ and $L'_{a_i}(x)$ are defined in Equation 3.
4. If the relation holds, then $\mathcal{B}_4$ outputs 0.

Otherwise, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_4$ first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly. It then queries its VPSS pseudorandomness oracle $\mathcal{O}^{\mathsf{Gen}}$ on input (k, y), receiving in return (d_k, D_k). It sets $r_k = d_k$ and $R_k = D_k$. $\mathcal{B}_4$ then outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, \mathsf{m}, C, \{(y_i, R_i)\}_{i \in C})$, $\mathcal{B}_4$ does the following:

1. Checks if $|C \cap \mathsf{corrupt}| < t$, if the check does not hold, $\mathcal{B}_4$ aborts.
2. Otherwise, $\mathcal{B}_4$ derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly.
3. It then checks that $y = y_i$ for all $i \in C$, and that $k \in C$. If any check fails, $\mathcal{B}_4$ outputs $\perp$.
4. It then queries its VPSS pseudorandomness oracle $\mathcal{O}^{\mathsf{Gen}}$ again on input (k, y), re-obtaining (d_k, D_k).
5. It then follows the remainder of the protocol honestly, deriving $z_k = r_k + c \cdot \mathsf{sk}^{(2)}_k$.
6. $\mathcal{B}_4$ then outputs z_k as its output.

Finally, when $\mathcal{A}$ terminates, $\mathcal{B}_4$ outputs 1.

Difference between Game 4 and Game 3. Because we assume $|\mathsf{corrupt}| < t$, $\mathcal{B}_4$ will abort with zero probability. The additional advantage to $\mathcal{A}$ in Game 4 is thus upper-bounded by the advantage that $\mathcal{B}_4$ wins its VPSS pseudorandomness game against Shine. In particular, when the pseudorandomness game bit $b = 0$, then $\mathcal{B}_4$'s simulation is identical to that of Game 3. and if $b = 1$, then it is identical to Game 4.

Let W_4 be the event that $\mathcal{A}$ wins in Game 4, and let $\mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{B}_4,\mathsf{Shine}}$ be the advantage that $\mathcal{B}_4$ wins the pseudorandomness experiment against Shine. Then

$$|\Pr[W_4] - \Pr[W_3]| \leq \mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{B}_4,\mathsf{Shine}} \leq \frac{q_1}{q} \tag{22}$$

where q_1 is the number of times $\mathcal{A}$ is allowed to query H_1.

Game 5. We now describe a simulating algorithm SIM that simulates Arctic to $\mathcal{A}$, with respect to a challenge group element $Y \in \mathbb{G}$ for which SIM does not know the corresponding discrete logarithm.

Setup. SIM accepts as input an instance X, which is a tuple consisting of public parameters $(\mathbb{G}, q, p)$, a discrete logarithm (DL) challenge $Y \in \mathbb{G}$, and the total number of parties n, the corruption threshold t, and the minimum number of signing parties μ. In addition, SIM accepts as input a set of $q_r = q_2 + q_3 + 2q_s + 1$ hash values $\{h_1, \ldots, h_{q_r}\}$.

Next, SIM initializes the following values:

- Initializes the table $\mathsf{Q_q} \leftarrow \emptyset$ to maintain signatures it issued for honest participants during its simulation, and the table $\mathsf{Q_m} \leftarrow \emptyset$ to maintain the set of messages queried by $\mathcal{A}$ to $\mathcal{O}^{\mathsf{Sign}_2}$.
- Initializes the tables $\mathsf{Q_1}, \mathsf{Q_2}, \mathsf{Q_3} \leftarrow \emptyset^3$ to manage random oracle queries and responses.
- Initializes the table $\mathsf{Q_y} \leftarrow \emptyset$ to manage state between signing queries.
- Initializes the counter $\mathsf{k_r} \leftarrow 1$ to track issued random oracle outputs.

SIM then runs $\mathcal{A}(\mathsf{par}, n, t, \mu)$. $\mathcal{A}$ outputs the set of corrupted parties $\mathsf{corrupt}$ such that $|\mathsf{corrupt}| \leq t - 1$, as well as its internal state state_A. SIM sets $\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$ and must reveal the secret keys of the corrupted parties to $\mathcal{A}$, which it does in the next step.

Simulating KeyGen. To simulate key generation where SIM is the trusted dealer, SIM does the following:

1. It sets $\mathsf{pk} = Y$.
2. To simulate key generation for the signing keys, SIM first picks $t - 1$ secret key shares for the corrupt parties $(\mathsf{sk}_i^{(2)})_{i \in \mathsf{corrupt}} \xleftarrow{\$} \mathbb{Z}_q$, and defines their corresponding public keys $\mathsf{pk}_i^{(2)} \leftarrow g^{\mathsf{sk}_i^{(2)}}$ for all $i \in \mathsf{corrupt}$.
3. SIM then derives the public key shares $\mathsf{pk}_j^{(2)}, j \in \mathsf{honest}$ for the remaining honest parties in the exponent, as follows:

$$\mathsf{pk}_j^{(2)} = (\mathsf{pk}^{(2)})^{L_0(j)} \cdot \prod_{i \in \mathsf{corrupt}} g^{\mathsf{sk}_i^{(2)} L_i(j)} \tag{23}$$

   where $L_i(x)$ is the i^{th} Lagrange polynomial defined by $\mathsf{corrupt} \cup \{0\}$.
4. To simulate key generation for the randomness keys, SIM simply performs key generation honestly, deriving $(\mathsf{sk}_1^{(1)}, \ldots, \mathsf{sk}_n^{(1)}) \xleftarrow{\$} \mathsf{Shine.KeyGen}(n, t)$.
5. SIM then sets $\mathsf{sk}_j = (\mathsf{sk}_j^{(1)}, \mathsf{sk}_j^{(2)}, \mathsf{pk})$, for $j \in \mathsf{corrupt}$, and $\mathsf{pk}_i = (\mathsf{pk}_i^{(2)})$, for $i \in [n]$.

Then, SIM runs $\mathcal{A}^{\mathcal{O}^{\mathsf{Sign}_1, \mathsf{Sign}_2}}(\mathsf{state}_A, \mathsf{pk}, \{\mathsf{pk}_j\}_{j \in [n]}, \{\mathsf{sk}_i\}_{i \in \mathsf{corrupt}})$, and responds to its oracle queries, as we describe next.

Simulating Random Oracles. To simulate $\mathcal{A}$'s random oracle queries, SIM simply performs lazy sampling. It programs H_1 with values sampled from its own random tape; however, it programs $\mathsf{H}_2, \mathsf{H}_3$ with values from its input $\{h_1, \ldots, h_{q_r}\}$, as follows.

- $\underline{\mathsf{H}_1}$: When $\mathcal{A}$ queries H_1 on inputs (ϕ_i, y), SIM does the following:
  1. Checks if $\phi_i = \hat{\phi}$, where $\hat{\phi}$ is such that $(\cdot, \hat{\phi}) \in \mathsf{sk}^{(1)}_{\mathsf{honest}} \setminus \mathsf{sk}^{(1)}_{\mathsf{corrupt}}$, and where $\mathsf{sk}^{(1)}_{\mathsf{corrupt}} = \cup_{i \in \mathsf{corrupt}} \mathsf{sk}^{(1)}_i$, and $\mathsf{sk}^{(1)}_{\mathsf{honest}} = \cup_{j \in \mathsf{honest}} \mathsf{sk}^{(1)}_j$.
  2. If this check holds, SIM aborts. We refer to this bad event as $\mathsf{BadEvent}_1$.
  3. Otherwise, SIM checks to see if $\mathsf{Q}_1[(\phi_i, y)] \neq \perp$, if the check holds, it returns $\mathsf{Q}_1[(\phi_i, y)]$,
  4. Otherwise, it samples $\gamma \xleftarrow{\$} \mathbb{Z}_q$ (using its own random tape).
  5. It then sets $\mathsf{Q}_1[(\phi_i, y)] = \gamma$.
  6. Finally, SIM returns γ.
- $\underline{\mathsf{H}_2}$: When $\mathcal{A}$ queries H_2 on inputs $(\mathsf{pk}, \mathsf{m})$, SIM first checks to see if $\mathsf{Q}_2[(\mathsf{pk}, \mathsf{m})] \neq \perp$, If so, it returns $\overline{\mathsf{Q}_2[(\mathsf{pk}, \mathsf{m})]}$. Otherwise, it sets $y \leftarrow h_{\mathsf{k_r}}$, and increments $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$. It then sets $\mathsf{Q}_2[(\mathsf{pk}, \mathsf{m})] = y$. Finally, SIM returns y.
- $\underline{\mathsf{H}_3}$: When $\mathcal{A}$ queries H_3 on inputs $(R, \mathsf{pk}, \mathsf{m})$, SIM checks to see if $\mathsf{Q}_3[(R, \mathsf{pk}, \mathsf{m})] \neq \perp$. If so, it returns $\overline{\mathsf{Q}_3[(R, \mathsf{pk}, \mathsf{m})]}$. Otherwise, it sets $c \leftarrow h_{\mathsf{k_r}}$, and increments $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$. it then sets $\mathsf{Q}_3[(R, \mathsf{pk}, \mathsf{m})] = c$. Finally, SIM returns c.

Simulating Signing Oracles. To simulate signing oracles, SIM does as follows:

- $\underline{\mathsf{Sign}_1(k, \mathsf{m})}$: When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on participant identifier k and message m, SIM does the following:
  1. If $k \notin \mathsf{honest}$, output $\perp$.
  2. Derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly.
  3. SIM then checks to see if $\mathsf{Q}_{\mathsf{y}}[y] \neq \perp$. If so,
     (a) SIM parses $(\mathsf{m}', z, c, R, (R_j, z_j)_{j \in [n]}) \leftarrow \mathsf{Q}_{\mathsf{y}}[y]$.
     (b) SIM then returns (y, R_k).
  4. However, if this is the first time that $\mathcal{A}$ has queried $\mathcal{O}^{\mathsf{Sign}_1}$ on m, SIM then samples $z \xleftarrow{\$} \mathbb{Z}_q$, and sets $c \leftarrow h_{\mathsf{k_r}}$. It then increments the counter $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$.
  5. SIM then derives $R \leftarrow g^z \cdot \mathsf{pk}^{-c}$.
  6. Then, using its knowledge of $(\mathsf{sk}^{(1)}_i, \mathsf{sk}^{(2)}_i), i \in \mathsf{corrupt}$, SIM deterministically derives each $(r_i, R_i, z_i), i \in \mathsf{corrupt}$, by first deriving
     $$(r_i, R_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}^{(1)}_i, y)$$
     and then deriving
     $$z_i \leftarrow r_i + c \cdot \mathsf{sk}^{(2)}_i$$
  7. SIM then derives each honest party's commitment $R_j, j \in \mathsf{honest}$ with respect to the joint commitment R defined in Step 5 and each corrupted party's $R_i, i \in \mathsf{corrupt}$, via Equation 24:
     $$R_j \leftarrow R^{L_0(j)} \cdot \prod_{i \in \mathsf{corrupt}} R_i^{L_i(j)} \tag{24}$$
     where $L_i(x)$ is the i^{th} Lagrange polynomial defined by $\mathsf{corrupt} \cup \{0\}$.

8. SIM then derives each honest party's response $z_j, j \in \mathsf{honest}$ via Equation 25:

$$z_j \leftarrow z \cdot L_0(j) + \sum_{i \in \mathsf{corrupt}} z_i \cdot L_i(j) \tag{25}$$

9. SIM sets $\mathsf{Q}_\mathsf{y}[y] = (\mathsf{m}, z, c, R, (R_j, z_j)_{j \in [n]})$.
10. SIM then checks if $\mathsf{Q}_3[(\mathsf{pk}, R, m)] = \perp$; if the check does not hold, then a bad event has occurred. We refer to this bad event as $\mathsf{BadEvent}_2$. In this case, SIM aborts. Otherwise, SIM programs $\mathsf{Q}_3[(\mathsf{pk}, R, m)] = c$.
11. SIM then outputs (y, R_k).

– $\underline{\mathsf{Sign}_2(k, \mathsf{m}, C, (y_j, R_j)_{j \in C})}$: When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on honest participant identifier k, message, coalition, and tuples $\{(y_j, R_j)\}_{j \in C}$, SIM does the following:

1. Honestly follows the protocol to ensure that $|C| \geq 2t - 1$, and returns $\perp$ if the check does not hold.
2. Checks to ensure that $k \in \mathsf{honest}$ and $k \in C$, if not, SIM outputs $\perp$.
3. Derives $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, \mathsf{m})$ honestly.
4. If $y_i \neq y'$ for any $i \in C$, then SIM outputs $\perp$.
5. SIM verifies the query is valid, by checking that $\mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j \in C}) = 1$. If the check does not hold, SIM outputs $\perp$.
6. SIM then checks if $\mathsf{Q}_\mathsf{y}[y'] \neq \perp$. If the check does not hold, then a bad event has occurred; we refer to this bad event as $\mathsf{BadEvent}_3$. In this case, SIM aborts.
7. Otherwise, SIM parses the entry $(\mathsf{m}, z', c', R', (R'_j, z'_j)_{j \in [n]}) \leftarrow \mathsf{Q}_\mathsf{y}[y']$.
8. If $(R_j)_{j \in C} \neq (R'_j)_{j \in C}$, SIM aborts. We refer to this event as $\mathsf{BadEvent}_4$.
9. Otherwise, SIM derives $R \leftarrow \prod_{j \in C} R_j^{\lambda_j}$, where λ_j is defined by C. If $R \neq R'$, SIM aborts. We refer to this bad event as $\mathsf{BadEvent}_5$.
10. Otherwise, SIM returns z'_k.

Analysis of SIM*'s Simulation.* The simulation of H_1, H_2, and H_3 are perfect because SIM simply simulates by lazy sampling.

SIM's simulation of key generation for signing keys with respect to pk for which it does not know the corresponding sk is perfect, because $\mathcal{A}$ is allowed to corrupt up to $t - 1$ participants. As such, SIM chooses $t - 1$ random secret keys $\mathsf{sk}_i^{(2)}, i \in \mathsf{corrupt}$, and then simulates signing for the (unknown) honest signing keys $\mathsf{sk}_j^{(2)}, j \in \mathsf{honest}$. SIM follows the protocol honestly when it derives Shine keys.

SIM's simulation of signing is indistinguishable from honestly following the protocol when SIM does not abort, because the output signature z'_k from $\mathcal{O}^{\mathsf{Sign}_2}$ is valid with respect to the relation $g^{z'_k} = R_k \cdot (\mathsf{pk}_k^{(2)})^c$, where R_k is output from $\mathcal{O}^{\mathsf{Sign}_1}$. Further, by Equations 24 and 25, all honest parties' commitments $(R_i)_{i \in \mathsf{honest}}$ and responses $(z_i)_{i \in \mathsf{honest}}$ are consistent with corrupted parties' commitments and responses for any signing session.

SIM aborts with negligible probability more than Game 4 because:

1. When SIM aborts due $\mathsf{BadEvent}_1$ and $|\mathsf{honest}| \geq t$, then Game 4 also aborts.
2. The probability that $\mathsf{BadEvent}_2$ occurs is q_3/q because it can only occur if $\mathcal{A}$ queries $\mathsf{H}_3(R, \mathsf{pk}, \mathsf{m})$ for the correct R for the given m as computed in step 5 of SIM's simulation of Sign_1.
3. SIM aborts due to $\mathsf{BadEvent}_3$ with zero probability, because we assume the use of authenticated channels.
4. If SIM aborts due to $\mathsf{BadEvent}_4$, then $|C \cap \mathsf{honest}| \geq t$, and so Game 2 also aborts.
5. If SIM aborts due to $\mathsf{BadEvent}_5$, then $|C \cap \mathsf{honest}| \geq t$, then both Game 3 and Game 4 would also abort.

Output. At the end of the game, $\mathcal{A}$ outputs a forgery $(m^*, \sigma^* = (R^*, z^*))$. If $g^{z^*} \neq R^* \cdot (\mathsf{pk})^{c^*}$ (meaning that the forgery is invalid), SIM outputs $\perp$. Without loss of generality, we assume that $\mathcal{A}$ queried H_3 on (R^*, pk, m^*).

Otherwise, if $\mathcal{A}$'s forgery is valid, SIM outputs the tuple (j, aux), where j is the index corresponding to $c^* = h_j$, and $\mathsf{aux} = (m^*, \sigma^*)$ is the adversary's forgery with respect to $\mathsf{pk} = Y$.

Difference between Game 5 and Game 4. The only additional advantage introduced in Game 5 is that SIM aborts on $\mathsf{BadEvent}_2$ with probability q_3/q if $\mathcal{A}$ queries H_3 before it is programmed. Let W_5 be the event that $\mathcal{A}$ wins in Game 4. Then

$$|\Pr[W_5] - \Pr[W_4]| \leq \frac{q_3}{q} \tag{26}$$

Finishing the Proof. The advantage of $\mathcal{A}$ against SIM is given by Game 5. Combining Equations 15, 16, 18, 20, 22, and 26 gives Equation 10. This completes the proof. □

Proof (Of Lemma 3). We now show that if there exists a PPT adversary $\mathcal{A}$ against the unforgeability of Arctic and an algorithm SIM as described in Lemma 2 that simulates Arctic to $\mathcal{A}$, then we can define a PPT adversary $\mathcal{D}$ that can efficiently solve for discrete logarithm challenges.

$\mathcal{D}$ begins by accepting a discrete logarithm challenge $Y \in \mathbb{G}$ and public parameters par. It then chooses a (t, μ, n) such that $n \geq \mu \geq 2t - 1$, and $\binom{n-1}{t-1} = \text{poly}(n)$.

$\mathcal{D}$ then executes the modified forking algorithm $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ as described in Section 3.3 and shown in Figure 1, providing as input the instance $X = (\mathsf{par}, Y, n, t, \mu)$.

$\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ then either outputs $\perp$ when it fails or the value (j, aux) when it accepts. Corollary 1 upper-bounds the probability that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ will output $\perp$ and Lemma 2 upper-bounds the additional advantage of $\mathcal{A}$ against SIM. Putting these two together gives Equation 27.[4]

$$\mathsf{accept}(\mathsf{Fork}_m^{\mathsf{SIM}}) \geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{A},\mathsf{Arctic}}(\lambda, n, t, \mu)^2}{q_r} - \frac{2(q_1 + q_3)}{q} - \frac{3q_r^2}{q} \tag{27}$$

We now show that when $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output (i.e., it does not output $\perp$), then $\mathcal{D}$ can solve for its discrete logarithm challenge Y with perfect success.

In the case that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output, it outputs the tuple $(h_j, h'_j, \mathsf{aux}, \mathsf{aux}')$. From these values, $\mathcal{D}$ then parses the following:

$$\begin{aligned} (m^*, \sigma^*) \leftarrow \mathsf{aux}, \quad (m^{**}, \sigma^{**}) \leftarrow \mathsf{aux}', \\ (R^*, z^*) \leftarrow \sigma^*, \quad (R^{**}, z^{**}) \leftarrow \sigma^{**}, \\ h_j = c^*, \quad h'_j = c^{**} \end{aligned}$$

Because h_j, h'_j are with respect to the same index j corresponding to $\mathcal{A}$'s random query for c^*, c^{**} in its first and second execution, then we know that $R^* = R^{**}$ and $m^* = m^{**}$. $\mathcal{D}$ can then solve for the discrete logarithm y to its challenge $Y = g^y$ by Equation 28:

$$y = \mathsf{sk} = \frac{z^* - z^{**}}{c^* - c^{**}} \tag{28}$$

Because Fork_m outputs $\perp$ in the case when any random oracle outputs collide (i.e., in the case when any $h_i = h'_i$), then we know that Equation 28 is solveable. This completes the proof. □

[4] Because SIM is run twice by Fork_m, the number of bad events that can occur is doubled.

D Details on Extending Shine and Arctic to be Robust

As presented, Shine is not a robust algorithm: if corrupt players output incorrect values from Shine.Gen, then Shine.Verify will compute that the $|C| \geq 2t-1$ commitments do not all lie on the same degree-$(t-1)$ polynomial in the exponent, and output 0. However, Shine.Verify will not necessarily be able to identify *which* party or parties misbehaved. This is because it is possible for the $t-1$ corrupt parties to output commitments that are wrong, but consistent with up to $t-1$ of the honest players' commitments, making it look like the remaining honest player was the outlier. Therefore, as we will see in the next section, Arctic will have to just abort in the case that Shine.Verify outputs 0.

However, we observe that in the *honest supermajority* case, where $|C| \geq 3t-2$, and so there are at most $t-1$ corrupt parties and at least $2t-1$ honest parties, Shine.Verify can be modified to be robust: if any corrupt parties output incorrect values from Shine.Gen, Shine.Verify will be able to notice the inconsistency *and identify the misbehaving parties*. Those parties can then be kicked out of C, and Arctic will be able to continue and produce its correct deterministic signature. The misbehaving parties may also be able to be kicked out of the system entirely, or have other external sanctions applied. The same observation was made by Cramer et al. [16] for their NIVSS, but in their setting, the *shares* are themselves published, whereas in Shine, only *commitments* are available to Shine.Verify.

The reason this works is based on a simple fact about polynomials, as used in error-correcting codes: there can only be at most one polynomial of degree $t-1$ that passes through at least $2t-1$ of a given set of $3t-2$ points. The reason is that if f_1 and f_2 each passed through a (possibly different) set of at least $2t-1$ of the $3t-2$ points, then those two sets must intersect in at least t points (by simple counting). Those t points uniquely define a polynomial of degree $t-1$, and so $f_1 = f_2$. In the honest supermajority setting, the at least $2t-1$ honest parties will give consistent outputs from Shine.Gen, and so there will be exactly one (not at most one) polynomial of degree $t-1$ that passes through at least $2t-1$ of the exponents of the $3t-2$ commitments.

The typical way to *find* this unique polynomial is with the Berlekamp-Welch algorithm, but Shine.Verify only has access to commitments to the purported polynomial evaluations, and not the evaluations themselves. However, we can take advantage of the fact that we are already assuming that $\binom{n-1}{t-1}$ is not too large, and take a more brute force approach. The modification to Shine.Verify is then as follows: if player k running Shine.Verify would output 0 in Step 4 of Shine.Verify, instead iterate through all $t-1$-sized subsets S of $C \setminus \{k\}$, and reconstruct (in the exponent) the polynomial that passes through player k's own point, and the points of S. If that polynomial fails to pass through at most $t-1$ of the points, those points are the misbehaving parties. Exclude them from C, and output 1.

There are $\binom{|C|-1}{t-1}$ possible subsets S, and at least $\binom{2t-1}{t-1}$ of those subsets contain only honest parties. If the iteration is done in a pseudorandom order (for example, keyed by d_k), then the expected number of iterations before the misbehaving parties are identified is at most $\binom{|C|-1}{t-1}/\binom{2t-1}{t-1}$, which is significantly smaller than $\binom{n-1}{t-1}$, which we already assumed is not too large.

Extending Shine to be robust will make round 1 of Arctic robust, so that the correct R can be computed even if some or all of the malicious parties submit incorrect R_j values. To make round 2 of Arctic also robust, we must also guard against an adversary submitting a correct R_j in round 1, but an incorrect signature share z_j in round 2.

In the event of the failure of the test $g^z \neq R \cdot \mathsf{pk}^c$ in Combine, one can identify the parties that submitted incorrect z_j values by checking $g^{z_i} \stackrel{?}{=} R_i \cdot (\mathsf{pk}_i)^c$. Once the parties submitting incorrect signature shares are eliminated, the remaining correct shares (of which there will be at least t) can be combined into the final signature $\sigma = (R, z)$.

D.1 Communication Model for Robust Arctic

For non-robust Arctic, we discuss in Remark 4 how our adversarial model requires authenticated channels, but does not assume replay protection. However, additionally in the robust setting, parties will require a communication channel that ensures protection against replayed messages, to ensure that an adversary

cannot replay old messages from honest participants, resulting in a potential abort. Furthermore, to achieve robustness, the communication model likewise requires liveness; the adversary cannot prevent honest parties from exchanging messages.